Ph.D. DISSERTATION

Fast and Cycle-approximate Simulation Techniques for Many-core NoC Architecture

매니코어 NoC 아키텍처에 대한 고속 사이클-근사 시뮬레이션 기법

FEBRUARY 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Shin-haeng Kang

Ph.D. DISSERTATION

Fast and Cycle-approximate Simulation Techniques for Many-core NoC Architecture

매니코어 NoC 아키텍처에 대한 고속 사이클-근사 시뮬레이션 기법

FEBRUARY 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Shin-haeng Kang

Fast and Cycle-approximate Simulation Techniques for Many-core NoC Architecture

매니코어 NoC 아키텍처에 대한 고속 사이클-근사 시뮬레이션 기법

지도교수 하순회

이 논문을 공학박사 학위논문으로 제출함

2016 년 10 월

서울대학교 대학원

전기.컴퓨터공학부

강신행

강신행의 공학박사 학위논문을 인준함

2016 년 12 월

위 원 장	최기영	(인)
부위원장	하순회	(인)
위 원	이재진	(인)
위 원	버나드 에거	(인)
위 원	오현옥	(인)

Abstract

Fast and Cycle-approximate Simulation Techniques for Many-core NoC architecture

Shin-haeng Kang
Department of Electrical Engineering and Computer Science
College of Engineering
The Graduate School
Seoul National University

Simulation is a software technique that uses the current available architecture to prototype a future architecture. In computer architecture research, simulation techniques are one of the most important skills. Simulation techniques enable us to obtain important performance indicators of new architectures and to perform the design space exploration using these metrics. Furthermore, the simulator enables rapid software development and optimization on the architecture that does not exist. Despite various known problems, such as slow speed or coverage issue, the reliance on simulation technology in computer architecture research continues to increase.

As the density of transistor increases and the performance improvement of the single core hits the ceiling, the newly constructed architectures usually consist of multi/many cores with the network-on-chip, which enables scalable communications. In addition, the implementation of the application itself has also been complicated to effectively utilize these parallel architectures. Thus, simulators for parallel architectures and parallel applications have become extremely complex, and existing sequential simulators no longer simulate these systems at a realistic time.

While many of parallel simulation techniques are being developed to solve these problems, they suffer from poor simulation performance or accuracy. In this thesis, we propose and evaluate a novel many-core simulation technique that can obtain the best simulation performance at the cost of minimum simulation error.

The proposed parallel many-core simulator is divided into three parts: 1) core simulator, 2) network-on-chip simulator, and 3) simulation backplane. Each core is executed by a core simulator, which communicates with the external simulation backplane via the Interprocess Communication (IPC). Each core simulation is performed individually in a separate host processor. The simulation backplane arranges messages from each core into chronological order, passes them to destination modules, and simulates hardware components other than cores. If the simulation backplane generates a request requiring NoC communication, this request is forwarded to the network simulator and is simulated at the most accurate accuracy level.

In this thesis, we proposed a novel core simulation model, which combined analytical and sampled simulations. The core simulator presents 11.36 to 44.31 MIPS performance, while the simulation error is approximately 8 percent. The standalone core simulator is released as an open-source.

We confirmed that NoC simulation has a great effect on the reliability of outputs generated from many-core simulation. First, existing flit-level NoC simulators were analyzed at source-code level. Based on the observations, various implementations were evaluated and various software optimizations was applied to improve the network simulation performance. The proposed NoC simulator presents more than 100KCycles/s performance unless the packet injection rate exceeds 0.00625, which is two times faster than state-of-the-arts NoC simulator at least.

The speed of the simulation backplane depends greatly on the IPC overhead and SystemC scheduling overhead. To reduce the IPC overhead, the trace-driven co-simulation technique is used, faster IPC is introduced, and the segmented L1 data cache is embedded in a core simulator. In addition, to reduce SystemC scheduling overhead, it is important to reduce the number of modules that are simultaneously awakened. To

this end, slave modules are redesigned to be activated only based on an event. A new scheduler parallelization technique is also studied. Although the newly developed SystemC parallel scheduler showed good performance under limited conditions, we also confirmed that no performance improvement was found in the TLM level many-core simulator developed in this thesis.

While the proposed many-core simulator uses the conservative synchronization technique which is free from causality errors and performs an accurate flit-level NoC simulation, the simulation performance is still acceptable, thanks to parallelism and optimizations. Additionally, the simulator is highly scalable to add other modules because the simulation backplane is developed to be compatible with SystemC TLM 2.0 standard. Although extensive experiments on accuracy are not conducted, it will be complemented when a detailed specification of the target architecture is given.

This dissertation can be a reference to the development of a many-core simulator, which will be more essential in the future.

Keywords: Many-core, Multi-core, Network-on-chip, Parallel Simulation, Virtual Prototyping, Synchronization

Student Number: 2010-20747

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Since manufacturing technology scales and the size of individual gates is reduced, physical limits of semiconductor-based microelectronics cause significant power dissipation and energy consumptions. As a promising way to seek further improvement, multiple independent processors are commonly used to increase a system's overall thread-level parallelism. A combination of increased available space and the demand for increased thread-level parallelism led to the development of many-core CPUs, which usually integrate a high number of cores (tens or hundreds). Such many-core architecture is aggressively applied to desktop computers, embedded systems, and hardware accelerators.

Network-on-chip (NoC) is a promising paradigm for on-chip communication. NoC borrows concepts and techniques from the well-established domain of computer networking and brings notable improvements over conventional bus and crossbar interconnections especially when the target architecture integrates a high number of components. NoC improves the scalability and the power efficiency of complex architecture compared to other designs.

An architecture simulator is an application that models computer devices or components. The main goal of the architecture simulator is to predict functional outputs and performance metrics on a given architecture and application. The architecture simulator has been the most popular way to evaluate hardware and software, playing an essential role in the computer architecture research. Table 1.1 classifies the performance evaluation methods for papers appearing in the International Symposium on Computer Architecture - the flagship conference in computer architecture - in six selected years [1]. In the conference's inaugural year, only two papers out of 28 (7.1 percent) were simulation-based, but that number steadily increased to 88 percent and 87 percent in 2001 and 2004, respectively. Despite the concern over the limited coverage of simulation-based evaluation, these statistics indicate that simulation is a critical component in computer architecture research. The main purposes of the architecture simulation are threefold: 1) performance evaluation for new architectures, 2) architectural exploration, and 3) early software development/optimization.

Many-core and NoC architecture impose enormous pressures on the simulation infrastructure as the architectural complexity and the number of processors increase in a system. Two technologies are essential for the practical simulation of many-core NoC architecture.

The first one is simulation modeling techniques. Since the number of concurrent heterogeneous components becomes tens or hundreds, it takes a lot of effort and time to model each of these simulation models. In addition, those components communi-

Table 1.1: Performance Evaluation Methodologies in Papers Appearing in the Proceedings of the International Symposium on Computer Architecture. The Table is Taken from [1].

Year	Total papers	Simulation	Measurement	Mathematical modeling	Other
2004	31	27	3	1	0
2001	25	22	2	0	2
1997	30	24	6	0	0
1993	32	23	9	6	1
1985	43	12	1	14	16
1973	28	2	0	5	21

cate very frequently, requiring a detailed network congestion model. Moreover, it is not a rare case that a detailed knowledge of its internal working, such as a latency of each operation, is not disclosed to public. Hence, it is also required to validate such simulation models by comparing with actual hardware. Worse yet, it is difficult to find the cause of the inaccuracy because of the complexity of the architecture. Therefore, it is often reasonable to reuse existing component simulators which are already validated by other researchers or component vendors. Since each component simulator may have different levels of abstraction and there are no unified interfaces between components simulator, it is important to establish common abstraction level, optimizations, and communication interfaces in order to use the existing component simulators. In this thesis, the open source emulator QEMU [2] is used as a functional simulator for each core, while another open source NoC simulator Noxim is used as a reference simulator for NoC simulation model. Meanwhile, it is also possible to apply modeling standard for portability and compatibility. In this thesis, each component is modeled complying with IEEE 1666 standard (OSCI TLM 2.0) [3].

The second essential technique is simulation performance acceleration technique. Since there are too many complex components to be simulated, traditional execution-driven sequential simulation techniques cannot exceed hundreds of KIPS performance. The simulation speed is sometimes prioritized over the simulation accuracy. One case is a microprocessor or system level design space exploration. To compare the performance of several candidate architectures, fast simulation is necessary to simulate all possible candidates. Another case is software development in the hardware/software co-design methodology. In this case, an application program should be executed repetitively for software development and debugging, so simulation speed is a crucial factor that affects the productivity of software design. Once function simulation and timing simulation are separated, faster timing model can be deployed. Note that such a fast timing model compromises a simulation accuracy to some extent. On the other hand, the parallel simulation is motivated from intuitive idea, which is aiming to accelerate many-core simulation by using multi-threaded implementation. However,

it is not actually intuitive to obtain the simulation performance gain through simulator parallelization. If a parallel simulation is not carefully designed, the performance would get worse. This phenomenon will be observed in the thesis. In addition, many of that research still suffer from scalability problem or allow some amount of timing inconsistency.

1.2 Contribution

In this research, we developed a fast and accurate parallel simulation framework for many-core NoC architecture. Performing conservative synchronization, no causality error is allowed. In addition, the dissertation focuses on three main components: core simulation, NoC simulation, and the simulation backplane.

- A core simulator executes an instruction stream, and generates a timed-event stream for communications between components. We proposed a fast and quite accurate abstracted timing model for a core simulation.
- A NoC simulator supports a detailed flit-level network simulation to model buffer- and link-congestions. We proposed and evaluated various sequential/parallel implementations and software optimizations. The proposed NoC simulator is at least two times faster than the state-of-the-arts.
- The simulation backplane contains a general simulation model for components of general many-core NoC architecture. The backplane aligns events from core simulators into chronological order. We presented multiple techniques to reduce the overhead of interprocess communications and the scheduling overhead. In addition, we presented a novel parallel distributed event scheduling engine, and showed its possibilities and limitations. The current simulation speed is up to 32 MIPS if the workload is well distributed to utilize all cores. The proposed simulator is competitive in comparison with existing state-of-the-arts in terms of accuracy and performance.

1.3 Dissertation Organization

The structure of the dissertation is as follows: The following section provides a brief overview of terminologies and existing many-core state-of-the-arts simulators. The detail descriptions of the core and NoC simulation are given in Section III and IV, respectively. The simulation backplane is followed in Section V. The study about the parallelization of the simulation backplane is given in Section VI. Finally, we summarized the proposed techniques and suggested some promising future works in Section VII.

Chapter 2

Background and Existing Research

2.1 Terminologies

2.1.1 Simulation Host / Simulation Target

Simulation host is the system executing simulation, and simulation target is the system being simulated. If arm system simulation is performed on x86 machine, x86 machine is the simulation host, and arm system is the simulation target. Therefore, many-core NoC architecture, we assume in this thesis, becomes the simulation target. On the other hand, if the simulation host and simulation target are not identical, an execution binary is generally incompatible. In this case, a cross-compiler is used to build a target binary using the host machine.

2.1.2 Simulated Time / Simulation Time

Simulation time is the time within the simulation host. Simulation time is consumed during the computation of the simulation (wallclock time). Simulated time is the time within the simulation target. Let us suppose that simulating 1 second of simulation target takes 1 hours of simulation host. In this case, the simulated time is 1 second, and the simulation time is 1 hour. If the simulation host and target are identical, the

simulator is 3600 times slower than the actual hardware.

2.1.3 User-level Simulation / Full-system Simulation

User-level simulation models a target microprocessor only. A user-level simulator can execute a cross-compiled binary image. On the other hand, full-system simulation models an entire computer system including a processor, an interconnection, a memory system, and I/O devices. A full-system simulator can execute a cross-compiled OS or load a disk image.

The most noticeable difference is the way to handle system calls. Since a user-level simulator executes a binary without an underlying target operating system, a target system call has to be passed to a host operating system, which will emulate the same functionality. While the functional accuracy is preserved, non-functional behaviors would be different between the simulator and the actual hardware. On the other hand, the full-system simulation is running a target operating system that actually can execute a system call, guaranteeing better accuracy.

Since scalable OS or ISA support beyond tens of cores is not available yet, many-core simulations are generally user-level for now. Hence, it is a common practice that user-level many-core simulators provide user-level virtualization to give user processes a virtualized system view.

2.1.4 Execution-driven Simulation / Trace-driven Simulation

The easiest way to distinguish between execution-driven and trace-driven simulation is to examine who is in the driver's seat: functional model or timing model.

Execution-driven or timing-driven simulation guarantees that the timing model is responsible for driving the functional simulation. An execution-driven simulator "runs" a execution binary. Hence, most cycle-by-cycle simulators belong to the execution-driven simulation technique. It is not easy to separate the functional model and the timing model from an execution-driven simulator

Trace-driven, event-driven or functional-first lets the functional model generate

time ordered record of events, and then the timing model "reads" and replay a trace of instructions captured during a previous execution. Hence, most instruction-by-instruction simulators belong to this category. It is easy to separate the functional model and the timing model from a trace-driven simulator.

Accordingly, an execution-driven simulator can fetch wrong path instructions from branch misprediction, while a trace-driven simulator cannot.

Nevertheless, trace-driven simulators attract many attentions thanks to its apparent separation of the function and the timing model, enabling the easy deployment of various timing models.

2.2 State-of-the-arts Many-core Simulators

In this section, six representative many-core simulators were reviewed. Most many-core simulators usually are modular-based, using existing simulators and libraries rather than develops from scratch.

2.2.1 Gem5

Gem5 simulation framework [4] is a combination of the M5 [5] and the GEMS [6] simulator. M5 is a simulator that simulates various ISA and CPU models, and GEMS is mainly a simulator for various memory systems and interconnect models. By combining them, Gem5 can simulate various systems from a single core to many-core. Gem5 also has a flexible modular structure, so it is convenient to add new modules or modify existing ones. Various simulation models on the speed-accuracy trade-off including the most accurate cycle-accurate model are supported, making it easy to select according to the purpose of the simulation. It is open to the public and has been actively developed until recently. Thanks to these advantages, Gem5 has already been used for performance evaluation in hundreds of publications, and it has been downloaded more than a thousand times. Many simulation papers are usually use Gem5 as the reference simulator. Gem5 uses an event-driven simulation engine to enhance simulation per-

formance, but it is still very slow. Parallelization has long been considered for faster simulation speed, but it is not yet complete. The precise flit-level network simulation module, called garnet [7], has been used for NoC simulation.

2.2.2 Marss

Marss [8] is a high-speed simulator that supports seamless switching between the cycle-accurate simulation mode and the native x86 emulation model. PTLSim [9] based on Xen hypervisor [10] is used for cycle-accurate simulation and QEMU[2] is used for native x86 emulation. The most important technology is that PTLsim and QEMU share the same CPU context structure, which enables a seamless transition between simulation and emulation mode. This functionality speeds up the simulation by rapidly passing through non-region of interest area. For the memory simulator, both a simple DRAM model and a cycle-accurate model using DRAMsim [11] are provided.

2.2.3 Sniper

Sniper [12] is a simulation framework that uses the x86 dynamic instrumentation tool Pin [13] for core simulation, and Graphite [13] for parallel simulation and many-core architecture. Pin can dynamically insert the instrumentation code into the running x86 code. By using this function, one of the analytical timing models [14]–[16], the interval core model[16], is inserted to perform timing simulation of each core. Graphite enables Sniper to simulate multi-program workload or multi-threaded shared-memory application in parallel. Since the interval-core model is very fast, and graphite greatly loosens synchronization between simulation components, Sniper presents very good simulation performance.

2.2.4 Zsim

Zsim [17] is a high-performance many-core simulator. For a core simulation, a detailed DBT-accelerated core model is also implemented on Pin [13]. The most impressive characteristic of Zsim is simulation performance, which can simulate a 1024-core sys-

tem at around 41 MIPS speed. The most important assumption in the synchronization scheme of Zsim is that when the two accesses are executed out-of-order, the order of the related events that occur is rarely changed. Based on this assumption, during a given interval/quanta (eg, 1000 cycles), each core generates traces as it proceeds without synchronizing. This phase is called the bound phase. Then, when the bound phase of all cores is over, parallel event-driven simulation is performed based on this trace to calculate the actual time that the events occurred. This phase is called the weave phase. However, since Zsim uses very naive network simulation model with fixed packet latencies, it is more likely to be error-prone in the network-intensive applications.

2.2.5 Manifold

Manifold [18] is a component-based parallel simulation framework capable of modular design. Manifold is a trace-driven simulator that allows a multithreaded, multicore emulator frontend to drive back-end timing models. The front-end emulator is implemented in QSIM [19] derived from QEMU. QSim instantiates independent QEMU CPU emulators for each guest thread. The actual timing simulation is performed through a timing backend that includes interacting simulation components. For parallel simulation, these components are allocated to logical processes, which are the units of parallel execution. For the core simulation, there are three options: the most precise cycle-accurate Zesto [20] for an out-of-order core based on Simplescalar [21], analytical SPX model for an in-order core, or k-CPI model.

2.2.6 Hornet

Hornet [22] is a parallel, highly configurable, cycle-level multicore simulator based on an ingress-queued worm-hole router network-on-chip (NoC) architecture. Hornet is originated from a parallel cycle-level NoC simulator DARSIM[23], and becomes a general multi-core simulator by using a built-in MIPS core simulator. Parallelization of NoC simulation also is implemented by adding two fine-grained locks in each virtual channel buffer, which is the only communication point in the two tiles, permitting con-

Table 2.1: Comparison of State-of-the-arts Many-core Simulators

	Scope	Timing Model	Parallelism	Network Model	Speed
Gem5[4]	User/Full	CA	Seq	Flit-level	<< 1 MIPS
Marss[8]	Full	Hybrid	Seq	Flit-level	<< 1 MIPS
Sniper[12]	Full	DBT+Instr.	Loose	Packet-level	2 MIPS
Zsim[17]	User	DBT+Instr.	Loose	Fixed-delay	41 (avg), 300 (max) MIPS
Manifold[18]	Full	Any	Cnsv/Loose	Flit-level	<< 1 MIPS
Hornet[22]	User	CA	Cnsv/Loose	Flit-level	Not available
Ours	User	DBT+Instr.	Cnsv	Flit-level	6 (avg), 32 (max) MIPS

* Abbreviations) CA: Cycle-accurate / Seq: Sequential / Anlt: Analytical / Samp: Sampled
Cnsv: Conservative, DBT+Instr: Dynamic binary translation + instrumentation

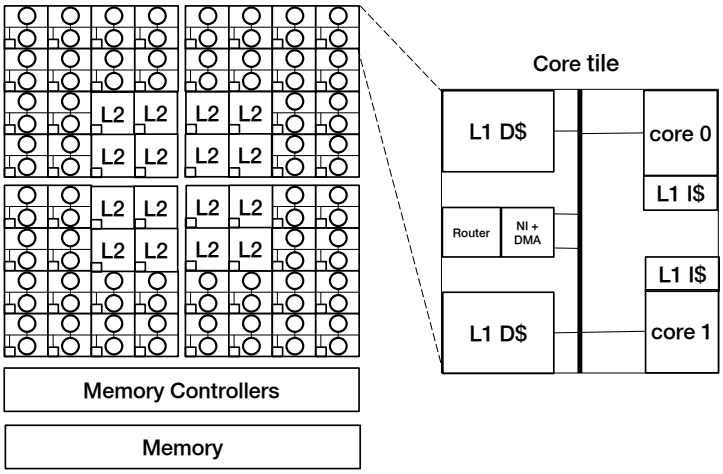
current accesses to each buffer by the two communicating threads. On the other hand, two synchronization schemes, cycle-accurate and periodic, are supported, allowing the user to trade-off between accuracy and performance.

2.2.7 Summary

The six simulators introduced here and our simulator are compared and summarized in Table 2.1. The simulation speed of various simulators are obtained from each paper. All papers was published after 2010, so we believe that simulation hosts do not make a big difference in the simulation performance.

Fast many-core simulators, like Sniper and Zsim, only support loose synchronization which would sacrifice the accuracy by allowing some amount of causality errors. Zsim and Sniper seem to abandon flit-level NoC simulation because it is too slow; Zsim deploys fixed-delay NoC model, and Sniper uses packet-level NoC simulation. Simulators supporting flit-level NoC simulation (Gem5, Marss, Manifold) shows only hundreds of KIPS speed.

In contrast, the proposed simulator guarantees better accuracy, supporting flit-level NoC simulation and conservative synchronization. In addition, the current simulation speed is up to 32 MIPS if the workload is well distributed to utilize all cores. Therefore, we believe that our simulator is an appropriate tradeoff between speed and accuracy, and is competitive in comparison with existing state-of-the-arts.



Parameter	Value
Tile configuration	48 core tiles and 16 L2 cache tiles
Core tile	2x Cortex-A15 @ 2.0Ghz
L1 I-cache	8KB 2-way set-associative, private for each core
L1 D-cache	8KB 2-way set-associative, shared for cores in each core tile
L1 cache block size	64 bytes
Cache coherency	Not supported by hardware
L2 cache tile	Static Non-Uniform Cache Architecture. 0.5MB 16-way set-associative.
L2 cache block size	64 bytes
NoC topology	8x8 2D mesh
NoC routing	XY
NoC link bandwidth	1 flit /cycle
NoC virtual channels per port	1
NoC virtual channel buffer size	32 flits
NoC sync period	cycle-accurate

Figure 2.1: Simulated Target Many-core System Characteristics

2.3 Host and Target Architecture

To guarantee the generality of the proposed simulation technique, we have chosen very general host and target architecture.

The detailed structure and specification of our simulation target are presented in Figure 2.1. The total number of cores is 96, satisfying the general definition of many-core.

We decided that the system does not support hardware-level cache coherence since

the coherency protocol overhead would exceeds the benefits of adding cores, which is known as the coherency wall problem. Instead, software-based coherency is supported.

On the other hand, the simulation host is a generic symmetric multiprocessor system (SMP) equipped with two Intel Xeon Processor E5-2640 v2. Each processor is octa-core with hyperthreading, so total 32 threads can be executed for the simulation. The main memory size is set to be enough for simulation, 32 GB.

Initially, we intended to use a cluster of SMPs, but we decided not to because excessive communication in many-core NoC architecture and high latency between different SMP hinder fast simulation.

Chapter 3

Core Simulation

3.1 Overview

Core simulation for performance evaluation can be divided into two main parts: functional simulation and timing simulation. Functional simulation model interprets and simulates instruction streams of target binaries, updating processor states such as register, and processor counter. The main purpose of the functional simulation is to generate correct program output which is identical to the output generated by the actual hardware. Most instruction-set-simulators support only functional simulation, so they cannot be used for performance evaluation. On the other hand, timing simulation provides the timing of simulated events. To this end, the timing model considers various micro-architectural structures. Therefore, it is extremely challenging to fulfill high-performance timing simulation.

In this dissertation, we propose a combined analytical/sampled timing simulation technique, which is independent of the specific functional simulator. As the processor timing model, we adopt one of representative analytical modeling techniques, the interval simulation technique [16]. Essential parameters in the formula are estimated through sampled simulation with a trace analyzer, while the other parameters are ob-

tained from architecture specification and functional simulation. As shown in Figure 3.1 with rectangle boxes, the structure of the proposed simulator consists of five components: functional simulator, branch predictor, memory hierarchy simulator, trace analyzer, and analytic performance estimator.

The base component of the simulator is the functional simulator. The functional simulator executes a cross-compiled target-machine binary on a host-machine. The functional simulator decodes target machine instructions, and sends minimum necessary information to the other components. The memory access information is transferred to the memory hierarchy simulator to detect the cache miss events. Branch instructions are sent to the branch predictor to detect the branch misprediction events. Sampled instruction streams are buffered to the trace buffer for the trace analyzer. The trace analyzer is invoked in a background thread only when the buffer is full, so that the time overhead of invoking the trace analyzer can be hidden. The analytic performance estimator calculates the time duration of the specific intervals according to the analytic formula, based on the parameter values collected from other simulator components. Each component is a modular, deployable, replaceable part of a simulator that could

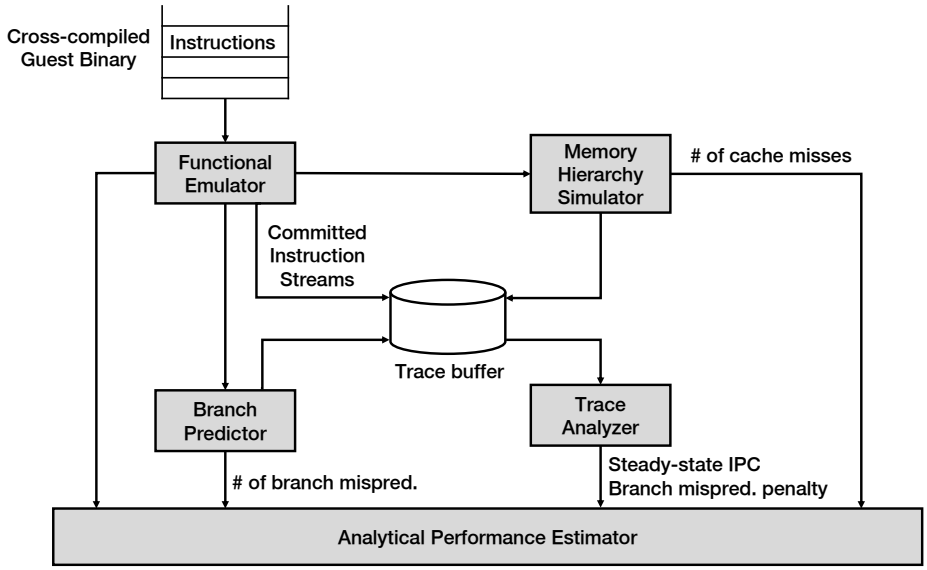


Figure 3.1: Structure of the Proposed Core Simulator

be configured to a specific processor by changing the parameter values.

The proposed technique is implemented over the popular and portable QEMU emulator, so named TQSIM (Timed QEMU-based SIMulator).

3.2 Related Works

3.2.1 Timing Models

- ***Cycle-accurate simulation*** validates a target architecture at the cycle level. The cycle-accurate simulator focuses on the accuracy to predict the performance of the target architecture precisely, updating values of all the state elements of the machine at every clock cycle. Unfortunately, such high accuracy comes at the price of high development cost and long simulation time; It is known that single-core cycle-accurate simulators typically run at 0.01 to 0.3 million instructions per second (MIPS) which means that it would take several days of simulation time to simulate a couple of minutes of simulated time. Moreover, the simulation speed is degraded even more as the architectural complexity and the number of processors increase in a system.

- ***k-CPI simulation*** assumes that it takes k cycles to execute one instruction. The number of cycles is given by one cycle for all instructions (1-CPI model) or given according to the datasheet of the simulated processor (datasheet model). The datasheet model uses the cycle counts for each instruction by consulting a table built upon the specifications datasheet from the simulated platform. Such k-CPI model is readily implemented on a top of the functional simulator without sacrificing the simulation speed. It is known that commercial processor models such as Imperas OVP [24] and ARM FastModels [25] essentially use the k-CPI approach. However, the credibility of timing estimation from the k-CPI model is very low, because the k-CPI model neglects complex behavior of modern microprocessor architectures.

- ***Analytical simulation*** estimates the processor performance by using mathemat-

ical formulas. It can be mainly classified into two approaches, based on the analytical performance modeling method: mechanistic modeling [14]–[16], and empirical modeling [26]. A mechanistic modeling constructs a model based on the mechanics of the target processor, called white-box modeling. Empirical modeling uses a parameterized performance model where parameters are decided by machine learning or regression analysis, without any specific knowledge about the micro-architecture of the target processor, called black-box modeling. The proposed model is a variant of the mechanistic model.

- ***Sampled simulation*** [27]–[29] performs cycle-accurate simulation with a number of sampling units rather than the entire instruction streams. The sampling units are selected either randomly [27], periodically [28], or based on phase analysis [29]. To guarantee that those sampling units successfully represent the whole application, the statistical methods would be applied. We adopt the idea of sampled simulation only to derive some of the parameter values which are required for the mechanistic formula of the analytical simulation technique.

- ***Statistical simulation*** [30], [31] generates short-running synthetic traces or benchmarks that are representative for long-running benchmarks, and uses them to speed up architectural simulation. Statistical simulation is composed of two phases. Statistics collection phase collects base program characteristics (basic instruction mix and instruction dependency) and micro-architectural dependent statistics (cache, branch statistics). This information is then used to generate a synthetic instruction trace that is fed to a simple processor model in the synthetic simulator phase.

- ***FPGA-accelerated simulation*** [32]–[34] implements timing models onto field-programmable gate-arrays (FPGA) to exploit fine-grain parallelism in the FPGA. It is possible that FPGA-accelerated simulation is used in conjunction with the software-based techniques, such as mechanistic model. FPGA-accelerated simulation demands additional hardware and development time to synthesize the

Table 3.1: Comparison of Various Modeling Techniques

	HW/SW	*Single Run	**Ref. HW/SW	Speed	Accuracy
Cycle-accurate	SW	O	O	Slowest	Best
k-CPI	SW	O	X	Fastest	Worst
Analytical	SW	O	X		
Sampled	SW	Case by case	O		
Statistical	SW	X	X		
FPGA-accelerated	HW	O	X		
Hybrid	SW	O	O		
Control-sensitive	SW	X	O		

* Single Run: is a single run enough to obtain a timing?
** Ref. HW/SW: does it require a reference hardware or software to give a timing information?

model into hardware.

- ***Hybrid simulation*** [8], [35], [36] uses bidirectional dynamic switching between a target cycle-accurate simulator and a functional simulator, while keeping the processor-centric state synchronized between both simulation modes. In HySim [35], [36], the target cycle-accurate simulator executes processor specific functions, whereas the host-compiled simulator executes target-independent parts of the application. Marss [8] supports seamless dynamic switching between the cycle accurate simulation mode and the native x86 emulation mode of QEMU. This switching mechanism speeds up program execution by skipping the parts which are of no interests to the users.

- ***Control-sensitive Simulation*** [37]–[39] makes use of context-sensitive estimates by keeping track of the execution history of the simulated target binary. Multiple execution times per basic block can be obtained at different contexts using a reference timing-accurate simulator [37] or static worst-case execution time analysis framework [38], [39] such as OTAWA [40] and Absint aiT [41]. The actual timing of basic blocks is defined dynamically depending on the previously executed basic blocks.

We have evaluated existent timing models as in Table 3.1: *k-CPI model* shows lack

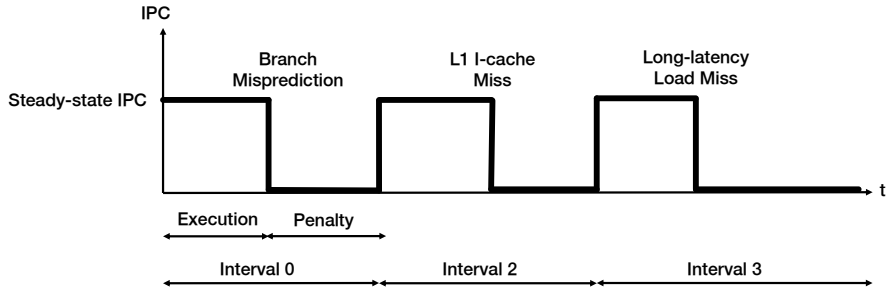


Figure 3.2: The Basic Idea of Interval Simulation: Execution Time is Partitioned into Discrete Intervals by Disruptive Miss Events such as Cache Misses and Branch Misprediction [15]

of accuracy; *sampled* and *hybrid simulation* require us to have a proper cycle-accurate simulator or to analyze the software statically; *control-sensitive* and *statistical simulation* usually should perform preprocessing of simulated code in advance. Therefore, we concluded that *analytical approach* is the best baseline technique. On top of that, we borrowed the philosophy of *sampled simulation* model, and devised the general scheduling analysis of sampled traces to improve the simulation accuracy of the analytical formula even further. The accuracy of *analytical model* is critically dependent on the accuracy of coefficient and parameters of the analytical formula. Existing techniques to obtain such parameters are inefficient, compromising the simulation speed of analytical techniques.

3.2.2 Analytical Model: Interval Simulation

Interval simulation [14], [16] has been proposed as the mechanistic modeling method, supporting out-of-order superscalar processors. In the interval analysis, execution time is partitioned into discrete intervals by disruptive miss events such as cache misses and branch misprediction as shown in Figure 3.2. It is based on two observations as follows:

- A superscalar out-of-order processor executes the number of instructions equal to a steady-state IPC (instruction per cycle), $SIPC$, at every clock cycle. Since the maximum number of instructions that enter the reorder buffer at every clock

cycle is the dispatch width, the dispatch width will be the steady-state IPC in the ideal case. However, the actual value is smaller than the dispatch width and how to compute the steady-state IPC is the key challenge of this method. The steady-state IPC is defined as the average number of committed instructions per clock cycle when cache accesses always hit, and all branches are correctly predicted. Then the total execution cycle becomes the number of committed instructions divided by the steady-state IPC under such ideal conditions.

- Disruptive miss events, such as cache misses and branch misprediction, interrupt the smooth flow of instruction execution. For example, when an instruction cache miss occurs, no more instructions are fetched until the cache miss is properly handled. Then cache miss penalty should be accounted for in the analysis.

Under these observations, the total execution cycle T is approximated as the following simple formula:

$$T = \frac{N_{total}}{SIPC} + \sum m_i \cdot p_i \tag{3.1}$$

N_{total} is the total number of simulated instructions, m_i is the number of disruptive events of type i, and p_i is the performance penalty of the event of type i. The disruptive events generally include L1 instruction cache misses, non-overlapping L2 cache misses, and branch mispredictions. Note that L1 data cache misses are not included. The performance penalty is determined based on the type of the event.

- **For instruction cache miss:** Instruction cache misses block the continuous inflow of new instructions until the miss event is handled. Hence, the first level instruction cache miss penalty becomes the performance penalty.

- **For data cache miss:** Data cache misses are divided into two categories [14]: short misses have latency significantly less than the maximum reorder buffer fill time, while long misses have latency significantly greater than the maximum reorder buffer fill time. Short miss does not block the instruction inflow to the

reorder buffer, so a short miss instruction is treated as a long-latency instruction in the architectural point of view. On the other hand, a long miss usually blocks the continuous instruction inflow to the reorder buffer, because the reorder buffer becomes full until the miss event is properly served. Hence, the performance penalty of a long miss often becomes the cache miss penalty. It is generally assumed that a first-level data cache miss is a short miss while a higher-level (second- or third-level) data cache miss is a long miss.

- **For branch misprediction:** Branch misprediction penalty becomes the sum of branch resolution time and the front-end pipeline depth where the front-pipeline length denotes the latency between instruction fetch and instruction dispatch. Once the mispredicted branch enters the instruction queue, no more useful instructions enter the instruction queue until the mispredicted branch is resolved. After the branch misprediction is detected and the correct branch target instruction is fetched, the pipeline is flushed and fetching begins from the correct path. The correct path instructions take front-end pipeline depth cycles to reach the instruction queue.

If several long misses occur in a short period of time, the corresponding miss penalties may be overlapped since multiple cache misses can be handled concurrently in the architecture. We identify the overlapped case by checking if the distance between the first and second cache miss events is smaller than the size of reorder buffer. Overlapped long cache misses are counted only once, thereby exposing memory-level parallelism (MLP). In contrast, consecutive branch mispredictions are not overlapped. Let us suppose that two consecutive branch mispredictions occur as an example of bursty branch mispredictions. In our model, the second mispredicted branch enters the instruction queue just after front-end pipeline refilling, and no more useful instruction enters the instruction queue until the second mispredicted branch is resolved. Thus, restoring IPC to the steady-state IPC is delayed again by the branch resolution time plus the front-end pipeline depth like the first misprediction. After all, bursty branch mispredictions do

not affect the corresponding branch misprediction penalties.

In the analytical formula, cache miss penalties are easily known when the architecture configuration is settled. The numbers of cache misses and branch mispredictions are obtained by a cache simulator and a branch predictor, which are attached to the functional simulator. However, the branch misprediction penalty varies depending on the situation when the mispredicted branch is fetched. The steady-state IPC also varies depending on programs and the ranges of interest even in the same program.

The simplest way to calculate the steady-state IPC might be to perform cycle-accurate simulation with perfect caches and a perfect branch predictor. Since this method requires us a cycle-accurate simulation, this approach is not appropriate for our purpose. Besides, a cycle-accurate simulator may not be available at the early stage of architectural design.

Another way to estimate the steady-state IPC is to use **IW** characteristics, based on the average functional unit latency and Little's law as proposed in an earlier work [14]. **IW** characteristic is a function that determines the number of instructions that **I**ssue in a clock cycle, given the number of instructions in the **W**indow. **IW** characteristic is represented by a curve $\mathbf{I} = \alpha \mathbf{W}^{\beta}/L$, where the values of α and β are specific to each benchmark, and L denotes the average instruction latency. However, estimating **W** without an accurate pipeline simulation is not trivial. Even worse, obtaining the values of coefficient α and β requires us the analysis of applications.

Another approach has been proposed to use the critical path length of the instruction stream to obtain the steady-state IPC [16]. Conceptually, a ROB-sized window slides along the dynamic instruction stream. Intuitively, the window cannot slide faster than the rate that the processor is issuing instructions belonging to the critical path, which is the longest data dependence chain for that window. Since finding the exact critical path length is time-consuming task, the authors of [16] also presented an approximation method to compute the critical path length in the window. Resource contention modeling for interval simulation has recently been proposed to consider the number of functional units [42].

In this dissertation, we propose to use a sampled simulation that determines the steady-state IPC. By using the in-house trace analyzer for a sampled window of instruction streams, we reduce the simulation performance loss, while maintaining the high accuracy of the steady-state IPC.

3.3 Sampling Mechanism

Using accurate steady-state IPC and branch misprediction penalty is essential to perform interval simulation. The steady-state IPC is affected by dependencies between instructions, execution time of instructions, and micro-architecture of a processor. Hence, obtaining the accurate steady-state IPC in a reasonable time is a challenging task.

Analysis of the whole instruction stream is against the philosophy of analytical simulation, which is designed to predict the performance by a simple formula. Moreover, the speed of such analysis is significantly slower than a functional simulator augmented with a cache simulator and a branch predictor. Existing studies of sampled simulation have demonstrated the potential of using representative sampled traces. Thus, we propose to use steady-state IPC and branch misprediction penalty obtained from the collected traces.

The proposed sampled simulation is different from conventional sampled simulation which depends on timing information of sampling units to predict the overall performance of an application. It is worth noting that we use sampled traces to estimate the steady-state IPC and the average misprediction penalty, so that the trace analyzer requires us a minimal abstract model to schedule the sampled instruction trace. Hence, the trace analyzer has been implemented in only several thousand lines of codes with a minimal development effort. Most of discrepancies between the trace analyzer and a cycle-accurate simulator stem from omitted or unknown processor details. In addition, the choice of sampling units does not have a significant effect on the accuracy of timing estimation of the proposed technique if the sampling unit has enough instructions to

warm up the window, which will be discussed with experimental results later.

3.3.1 Sampling Configuration

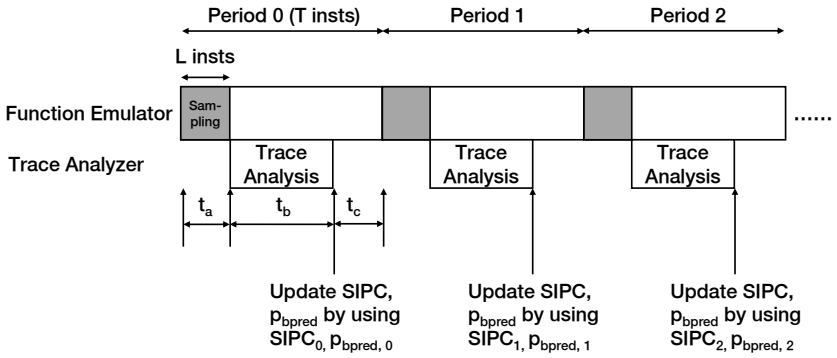
In sampled simulation, the sampling units are selected either randomly, periodically, or based on phase analysis. Among them, we use periodic sampling, which is the simplest form to implement. Hence, a sampling configuration (L/T) is specified as the sampling size L and the sampling period T in terms of the number of instructions, and the sampling duty cycle is given by $\mathrm{L/T} \times 100(\%)$. Once the specified number of instructions is collected in the buffer as the functional simulator proceeds, the trace analyzer is invoked for the buffer as a separate thread. The overhead of executing the trace analyzer is to some degree hidden by the use of an independent thread. This approach minimizes the overhead of running the trace analyzer, yet achieving reasonably accurate timing result. The overall simulation progress is described in Figure 3.3.

Excessively high duty cycle slows down the total simulation time, without improving the accuracy any more. Although the trace analysis is performed in a separate thread, we maintain only one trace buffer. Once the duty cycle increases after a certain breakpoint and $t_a + t_b$ is longer than the sampling period, the functional simulator has to be blocked until the trace analyzer processes all instructions in the trace buffer as described in Figure 3.3(b). Therefore, the simulation time dramatically increases after the breakpoint.

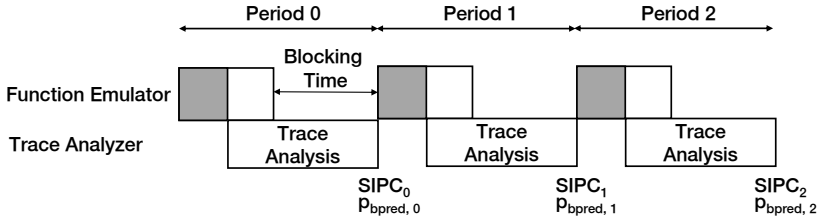
We should consider the trade-off relationship between the simulation speed and the accuracy when determining the L and T values, which will be discussed with experimental results in the next section.

3.3.2 Parameter Extraction

The analytical formula of the simulator uses the mean values of steady-state IPCs and branch misprediction penalties of sampled trace. Suppose that the application has n sampling periods $\{0, 1, \ldots, n-1\}$. Let N_i and $m_{bpred,i}$ the number of instructions and mispredicted branches that belong to i-th sampling period, respectively. If the steady-



(a) How to Make Use of Steady-state IPC and Branch Misprediction Penalty Obtained from the Analysis of the Sampled Trace



(b) QEMU is Blocked until the Trace Analyzer Processes All Instructions in the Trace Buffer

Figure 3.3: Simulation Progress with the Trace Analyzer If L Instructions are Collected Periodically Every T Instructions

state IPC and the branch misprediction penalty of i-th period are given as $SIPC_i$ and $p_{bpred,i}$, the mean values of steady-state IPCs and branch misprediction penalties are defined as follows:

$$SIPC_{mean} = \frac{n}{\frac{1}{SIPC_0} + ... + \frac{1}{SIPC_{n-1}}} \tag{3.2}$$

$$p_{bpred,mean} = \frac{p_{bpred,0} \cdot m_{bpred,0} + ... + p_{bpred,n-1} \cdot m_{bpred,n-1}}{m_{bpred}} \tag{3.3}$$

$SIPC_{mean}$ is the harmonic mean of $SIPC_0, SIPC_1, \ldots, SIPC_{n-1}$; $p_{bpred,mean}$ is the weighted arithmetic mean of $p_{bpred,0}, p_{bpred,1}, \ldots, p_{bpred,n-1}$ where each weight is determined by the ratio of the number of mispredicted branches in the sampling period to the total number of mispredicted branches.

Given analytical formula (3.1), $SIPC_i$ and $p_{bpred,i}$ of the i-th sampling period, the

simulated cycles of n periods can be calculated as [1]:

$$T = \frac{N_0}{SIPC_0} + m_{bpred,0} \cdot p_{bpred,0} + \ldots \frac{N_{n-1}}{SIPC_{n-1}} + m_{bpred,n-1} \cdot p_{bpred,n-1} \quad (3.4)$$

As we use a fixed sampling length and period, $N_0 = ... = N_{n-1} = N_{total}/n$ holds. Let us combine $(N_i/SIPC_i)$-type terms in equation (3.4):

$$\begin{aligned} T' &= \frac{N_0}{SIPC_0} + \frac{N_1}{SIPC_1} + \ldots + \frac{N_{n-1}}{SIPC_{n-1}} && (3.5)\\ &= \frac{N_{total}}{n \cdot SIPC_0} + \frac{N_{total}}{n \cdot SIPC_1} + \ldots + \frac{N_{total}}{n \cdot SIPC_{n-1}} && (3.6)\\ &= \frac{N_{total}}{n}(\frac{1}{SIPC_0} + \ldots \frac{1}{SIPC_{n-1}}) && (3.7)\\ &= \frac{N_{total}}{SIPC_{mean}} && (3.8) \end{aligned}$$

On the other hand, combining $(m_{bpred,i} \cdot p_{bpred,i})$-type terms in equation (3.4) gives the following result:

$$\begin{aligned} T'' &= m_{bpred,0} \cdot p_{bpred,0} + \ldots + m_{bpred,n-1} \cdot p_{bpred,n-1} && (3.9)\\ &= \frac{m_{bpred,0} \cdot p_{bpred,0} + \ldots + m_{bpred,n-1} \cdot p_{bpred,n-1}}{m_{bpred}} \times m_{bpred} && (3.10)\\ &= p_{bpred,mean} \cdot m_{bpred} && (3.11) \end{aligned}$$

Hence, equation (3.4) can be rewritten:

$$T = \frac{N_{total}}{SIPC_{mean}} + p_{bpred,mean} \cdot m_{bpred} \quad (3.12)$$

It implies that we may use the mean values to obtain the simulated cycle.

Concurrent execution of the functional simulator and the trace analyzer enables

[1] Additional simulated cycles incurred by cache misses are not affected by the steady-state IPC or branch misprediction penalty, so we can ignore them here

faster simulation. However, the simulated cycle at a specific point of time has to be calculated based on parameters which do not include results of the current period. We illustrated this situation in Figure 3.3(a). If the sampling period is $t_a + t_b + t_c$ seconds, the analytical formula should use old parameters during $t_a + t_b$ seconds, possibly degrading the accuracy of the simulated cycle estimated from the proposed technique. Furthermore, the point in simulated time when the updated parameters for the analytical model become available depends on the execution time of the trace analyzer. It would make the approach non-deterministic (results vary between runs on the same hosts) and simulation results depend on the performance of the simulation host (results vary between different hosts). However, we believe that its impact is not significant, compared with the accuracy error caused by the other factors. Non-determinism would be minimized if the simulator exclusively uses a homogeneous SMP (Symmetric multiprocessor system), which is the common simulation environment. The worst-case scenario will occur when parameters of the current period are not available until simulating all instructions in the period is completed. Note that its impact is like the situation that we double the sampling period. Experiments confirmed that once a sampling duty cycle exceeds a certain point (≥ 0.001), the sampling duty cycle is mostly irrelevant to the simulation accuracy.

3.4 Trace Analyzer

Instead of modeling a specific micro-architecture of a processor, we developed a trace analyzer for generic out-of-order superscalar processors. We assume that the target out-of-order processor is well balanced so that as many instructions as the dispatch width can be put into the window at every clock cycle if the reorder buffer has available space and the cache miss or branch misprediction does not occur. These conditions are generally satisfied with the sufficient size of fetch, decode, rename width.

The trace analyzer focuses on the flow of instructions from the dispatch stage of a superscalar out-of-order processor; it models dispatch, issue, writeback, and commit stages only, while abstracting out fetch, decode, and rename stages. The core struc-

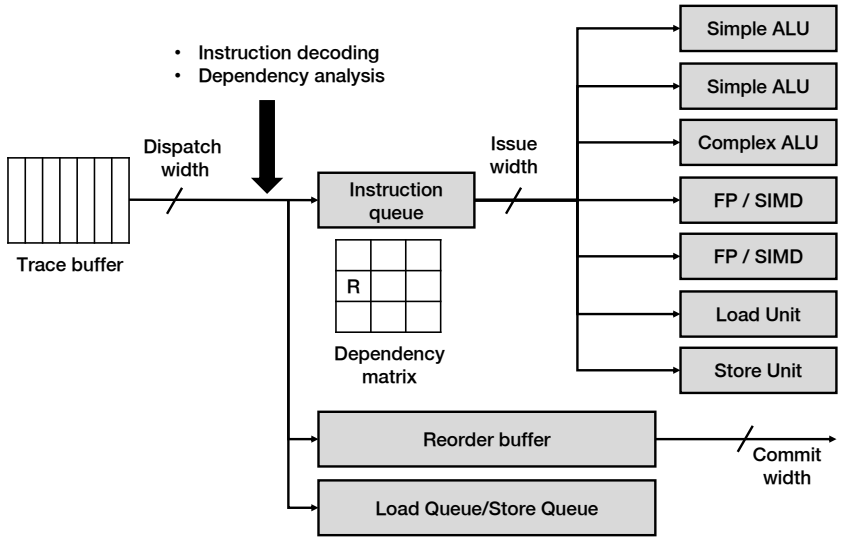


Figure 3.4: Core Structure of the Trace Analyzer when it is Configured for ARM Cortex A-15

Obtained from the functional simulator augmented with the cache and branch simulator				Obtained after instruction decoding			
Machine Code	Effective Memory Address	L1 Data Cache Miss	Branch Prediction Correctness	Instruction Type	Target Functional Unit	Source Registers	Target Registers
0a000001			Correct	Branch	SimpleALU	Cond. Flags	
e0854004				IntALU	SimpleALU	R4, R5	R4
e59f4064	0x1fac8	Y		MemRead	LoadUnit	PC	R4

Figure 3.5: Internal Structure of the Trace Buffer

ture of the trace analyzer is illustrated in Figure 3.4. Since the trace analyzer uses the trace generated from a functional simulator, it needs not be concerned about functional correctness.

After an instruction is executed by the functional simulator that is augmented with the cache and branch simulator, a new trace entry is added to the trace buffer queue. The trace analyzer waits until the trace buffer is full. Three example entries in the trace buffer are illustrated in Figure 3.5. Left four columns of the table depict the entries (machine code, effective memory addresses, L1 data cache miss, and branch prediction correctness) that are obtained from functional simulation.

When an instruction in the trace buffer is dispatched into the instruction queue

and the reorder buffer, the trace analyzer decodes the machine code of the instruction; so, the right four columns of the table (instruction type, target functional units, source register, and target register) can be filled. At the same time, we perform the preliminary dependency checking.

How to use this information in the trace buffer is summarized as follows:

- *Source register and target register:* to find the register dependency.
- *Effective address:* to find the memory dependency.
- *L1 data cache miss:* to add an additional latency of the instruction which caused L1 data cache miss. Recall that a short miss instruction is treated as a long-latency instruction in the architectural point of view. Consequently, the effect of L1 data cache misses is included in the steady-state IPC.
- *Branch prediction correctness:* to track the branch resolution time of the instruction that makes a misprediction.
- *Instruction type and target functional unit:* to issue the instruction to the associated functional unit.

Once all instructions in the trace buffer are processed by the trace analyzer, the steady-state IPC and branch misprediction penalty of the specific period are derived: the steady-state IPC is the value of the sampling length (in the number of instructions) divided by cycles required to process the trace buffer; the branch misprediction penalty is the value of the accumulated misprediction penalties divided by the number of mispredicted branches throughout the specific period. Those values are used to calculate the mean values of steady-state IPC and misprediction penalty for the entire application.

3.4.1 Dependency Analysis

The important technique involved in the trace analyzer is to manage dependency between instructions. In addition to data dependencies, we should consider control and

Reorder buffer

Idx	0	1	2	3	4	5	6	7	8	9
	inst_8	inst_9	inst_0	inst_1	Inst_2	Inst_3	Inst_4	Inst_5	Inst_6	Inst_7

Relationship between Instructions

Register Dependency
inst_0 (2)-> inst_2 (4)
inst_6 (8)-> inst_8 (0)
inst 5 (7)-> inst_9 (1)

Control Dependency
inst_3 (5)-> inst_7 (9)

Memory Dependency
inst_2 (4)-> inst_8 (0)

Dependency Matrix

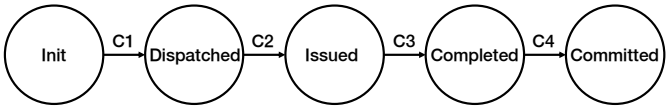
	0	1	2	3	4	5	6	7	8	9
0										
1										
2					R					
3										
4	M									
5										C
6										
7		R								
8	R									
9										

Figure 3.6: Dependency Matrix

memory dependencies carefully. We define a dependency matrix to represent the relationship between instructions.

The reorder buffer is a circular queue which contains all in-flight instructions, namely, all instructions that have been dispatched but have not yet committed. We use the fact that the index of an instruction in the reorder buffer is unique and invariable once the instruction is allocated to the reorder buffer. Suppose that the size of the reorder buffer is W_{rob}. Then the dependency matrix is a square $W_{rob} \times W_{rob}$ matrix M such that $M[i, j]$ is 'R' when there is a register dependency from the i-th instruction to the j-th instruction in the reorder buffer. 'C' and 'M' are used to denote control and memory dependency, respectively. At the point of dispatching an instruction, the initial dependency analysis of the instruction is conducted to fill the dependency matrix. The dependency matrix is, later, used to look up the state of previous instructions that the current instruction depends on.

Figure 3.6 illustrates the simple dependency matrix when $W_{rob} = 10$ and $inst_i$ is older instruction than $inst_j$ if $i < j$. For example, register dependency from $inst_0$ to $inst_2$ is expressed as $M[2, 4] =$ 'R'.



- C1: The remaining dispatch width is available / ROB & IQ & LDQ & STQ have an available buffer slot
- C2: The remaining issue width is available / all types of dependencies are met / the functional unit is available
- C3: The execution time is elapsed
- C4: The remaining commit width is available / the instruction is the oldest instruction in the ROB

Figure 3.7: Life Cycle of an Instruction. The Set of Transition Conditions is Associated with Each Edge

3.4.2 Life Cycle of An Instruction

The life cycle of an instruction can be specified by a finite-state machine (FSM) as shown in Figure 3.7. Each instruction may have five states from the set {*Init, Dispatched, Issued, Completed, Committed*}. An instruction is initially in *Init* state. If an instruction in the *Init* state is allocated to the instruction queue and the reorder buffer, the state of the instruction becomes *Dispatched*. If an instruction in *Dispatched* state becomes executable and the corresponding functional units are available, the instruction is issued to the corresponding functional unit, entering the *Issued* state. Instructions of *Issued* state are immediately removed from the instruction queue. The number of instructions issued per cycle may not exceed the maximum issue width. Once the execution time of the issued instruction elapses, the instruction is claimed completed and released from the functional unit, moving to the *Completed* state. As many completed instructions as the commit width may leave the window if they are the oldest instructions in the window. Instructions of *Committed* state are immediately removed from the reorder buffer.

Table 3.2: Simulated Target System Characteristics

Parameter	Value
CPU Codename	Cortex-A15
ARM ISA	ARMv7-A (32-bit)
Core clocks	2.0 GHz
Front-end width	3
Back-end width	8
Front-end pipeline depth	12
Branch predictor	Bi-Mode
IQ enties	48
LSQ entries	16 each
ROB entries	60
Functional Units	2 simpleALU, 1 complexALU, 2 FP/SIMD unit, 1 load unit, 1 store unit
L1 I-cache	32KB 2-way set-associative, 64B lines
L1 D-cache	32KB 2-way set-associative, 64B lines
L2 cache	unified, 1MB, 16-way set-associative, 64B lines, 10 cycle access time
Main memory	100 ns access time

Table 3.3: Benchmarks Description with Input Sets and Ratios (%) of Each Instruction Type

Benchmark	Configuration	Arith(Short)	Arith(Long)	Branch	Memory
basicmath	small	83.49	1.24	14.54	15.17
bitcnts	75000 items	90.90	0.00	13.62	9.10
qsort	small	63.84	0.14	19.78	36.02
susan	input_small.pgm -s	55.25	16.60	10.53	28.12
jpeg	-dct int -progressive -opt	57.84	1.39	11.17	40.77
dijkstra	small	78.48	0.07	22.76	30.86
patricia	small.udp	67.45	0.53	17.48	32.00
ispell	-a tests/small.txt	64.23	0.08	15.77	35.69
stringsearch	large	69.89	0.00	20.14	30.11
rijndael	input_small.asc	63.73	0.11	6.39	36.16
sha	input_small.asc	74.56	0.00	5.70	25.44
fft	4 4096	78.59	1.73	16.54	19.58
adpdm	data/small.pcm	84.25	0.00	6.86	15.75
gsm	-fps -c data/small.au	45.44	12.44	7.96	42.12

3.5 Experimental Results

Simulation target system: To evaluate the proposed technique, we first configure the simulation target system based on the Cortex A15 processor as closely as possible. Table 3.2 shows the system characteristics with key parameter values. By modifying some parameters, we will evaluate the adaptability of the proposed simulation technique, compared with the reference simulator in terms of accuracy.

Reference simulator / simulation error: For the reference simulator to compare the accuracy and speed of the proposed simulator, we chose a state-of-the-art open-source cycle-accurate simulator Gem5 [4]. Using open-source cycle-accurate simulator as a reference simulator has two advantages: 1) It is easy to identify the source of simulation errors since the reference simulator provides important system statistics; 2) It is easy to change the parameters of the system and CPU. If we use only one microarhictecture, it is easy to tune a simulator or a model for the target microarhictecture. It is reported that the maximum error of Gem5 is 15% with some exceptions in the SPEC benchmarks (gems, milc, namd) [43]. Therefore, we decided that comparison with Gem5 is a good starting point for simulation development.

We use the number of instructions that execute per cycle (IPC) to evaluate the overall accuracy of the proposed simulation. We define the IPC error:

$$IPC\ error = \frac{IPC_t - IPC_r}{IPC_r} \tag{3.13}$$

IPC_r is the reference IPC measured by the reference simulator; IPC_t comes from TQSIM. The average absolute IPC error is also defined as:

$$Average\ absolute\ IPC\ error = \frac{1}{n}(\sum^{n} \| \frac{IPC_t - IPC_r}{IPC_r} \|) \tag{3.14}$$

, where n is the number of benchmarks. In the context of design space exploration, the accuracy of speedup/slowdown prediction is also important, which indicates the ability of a simulation technique to predict how well our processor model tracks the results of detailed simulation by the reference simulator. Hence, we define the prediction error as the relative error of the predicted speedup/slowdown:

$$Prediction\ error = (\frac{IPC_{t,B}}{IPC_{t,A}} - \frac{IPC_{r,B}}{IPC_{r,A}})/(\frac{IPC_{r,B}}{IPC_{r,A}}) \tag{3.15}$$

$IPC_{t,B}$, $IPC_{t,A}$ denotes the IPC values obtained from TQSIM when running on two different microarchitectures A and B; $IPC_{r,B}$, $IPC_{r,A}$ are the similarly defined

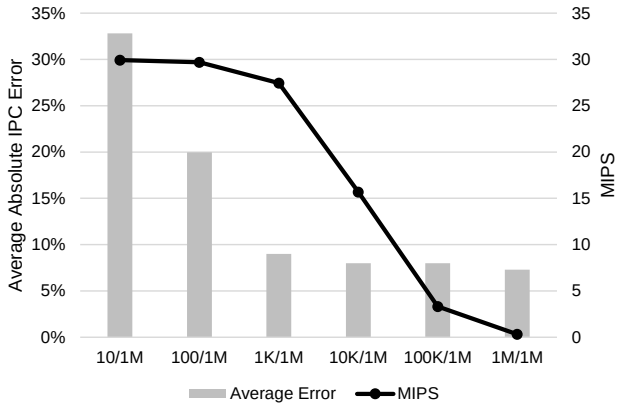
IPCs for the reference simulator. This definition quantifies the degree of difference in IPC increase/decrease obtained from TQSIM versus the IPC increase/decrease obtained from the reference simulator.

Benchmark application: We use 14 of the MiBench benchmarks [44]. See Table 3.3 for more details on these applications and the inputs that we have used. The frequency of occurrence of each instruction type is given to represent the application characteristics. The application selection procedure is based on the following steps: First, we choose only one application among applications with similar code characteristics. For example, we use only one application between FFT and IFFT applications. Second, we remove applications which failed to be built on our host system. Third, we remove applications which failed to be simulated with Gem5. Finally, we have 4 automotive, 2 networking, 1 consumer, 2 office, 2 security, and 3 telecom applications as the benchmark set. We believe that 14 applications and 5 different system configurations, 70 combinations, are enough to validate the generality of the proposed approach.

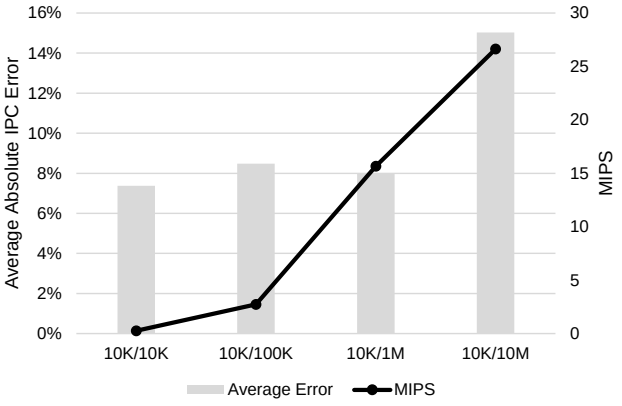
Simulation environment: All guest binaries were cross-compiled with *arm-linux-gnueabi-gcc* tool-sets for the ARM processor. The simulation host machine is equipped with an Intel i7-3770K processor clocked at 3.50GHz, 32 GB main memory, and Ubuntu Linux 64bit.

3.5.1 Time-accuracy Trade-off

First, we explore various sampling configurations to find the best compromise between the simulation error and the simulation speed. To separate the impact of the sampling size from that of the sampling period, we fix the sampling period as one million instructions, and vary the sampling size from ten to one million instructions. The simulation accuracy and speed are plotted in Figure 3.8(a). Increasing the sampling size results in lower error by having more instructions to be analyzed. From the experimental results, we observed that the average simulation error is stabilized around 8% after 1,000 in-



(a) Various Sampling Sizes and the Fixed Sampling Period (One Million Instructions)



(b) Various Sampling Periods and the Fixed Sampling Size (10,000 Instructions)

Figure 3.8: The Time-accuracy Trade-off: Fixed Sampling Period or Sampling Size

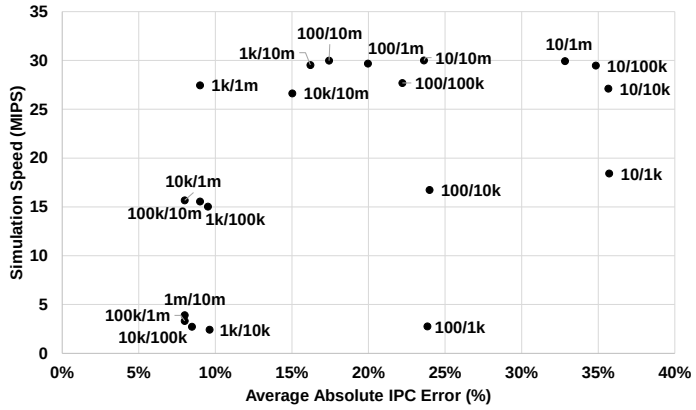


Figure 3.9: Simulation Error and Time Controlled by the Sampling Sizes and Sampling Period

structions. Inevitably, the window is empty when each trace analysis begins with the sampled instruction trace. Hence it is required to warm up the window to estimate the steady-state IPC in the trace analysis. The estimation error becomes more significant when the length of sampled instructions is too short to warm up the window. Meanwhile, the simulation speed decreases slowly until 1,000 sampling size, after which it decreases rapidly. Such a rapid drop is attributable to the congested trace buffer. The trace analysis becomes the speed bottleneck since writing new instructions to the trace buffer is blocked until the previous analysis finishes.

To measure the impact of various sampling periods, we use the constant sampling size, 10,000 instructions, and vary the sampling period from 10,000 to 10 million. Figure 3.8(b) shows that a long sampling period does not necessarily cause the accuracy loss until it becomes too large. Unlike the sampling size, the simulation accuracy is not sensitive to the sampling period unless the number of sampled traces is too small or the application's IPC characteristic changes too much. On the other hand, the simulation speed is significantly degraded if sampling the instruction trace is too frequent such as (10K/10K) and (10K/100K).

We tested all possible combinations of sampling periods and sizes in Figure 3.9. The plot indicates that sampling configuration (1K/1M) or (10K/1M) results in the best

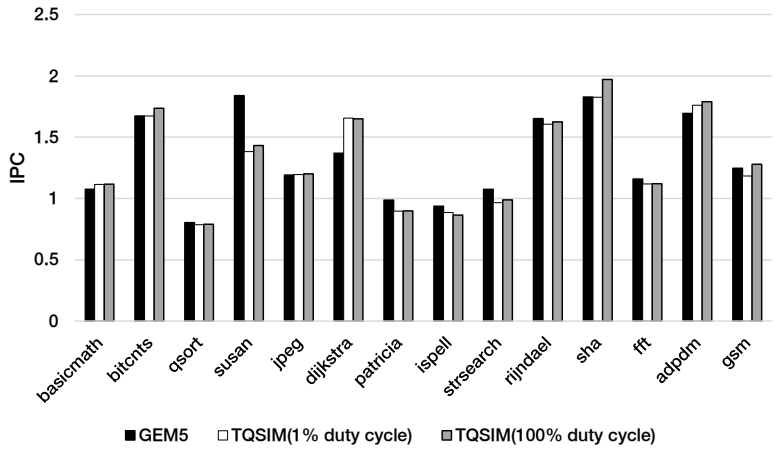


Figure 3.10: IPC (instructions Per Cycle) Obtained from Gem5 and TQSIMs

compromise between simulation time and accuracy for the given experimental setting and the benchmark applications.

We observed that the breakpoint is around 0.1-1% duty cycles, where the simulation time dramatically increases. It is mainly due to the trace buffer blocking described in Figure 3.3(b). To mitigate the blocking penalty, we may maintain multiple trace buffers in order to overlap multiple trace analyses. However, it will not improve simulation accuracy.

3.5.2 Simulation Accuracy

To validate the accuracy of TQSIM, we first validate the trace analyzer in isolation by changing the duty cycle of sampling from 100% duty cycle (sample always) to sparse sampling (1%). We measured the total simulated cycles and the overall IPCs obtained from TQSIM and Gem5. The result is shown in Figure 3.10. TQSIM with 100% duty cycle gives a simulation cycle error of -17.00% - 28.44%, with an average absolute error of 7.31 %. We believe that this error is acceptable since we model the architecture at a very high level unlike general cycle-accurate simulators usually do. For reference, the error of 1-CPI approach was measured to have 36.23% on average and 83.77% at maximum. It is also seen that IPC differences from duty cycle 100% and 1% represent a negligible amount, which justifies, to some degree, our sampling

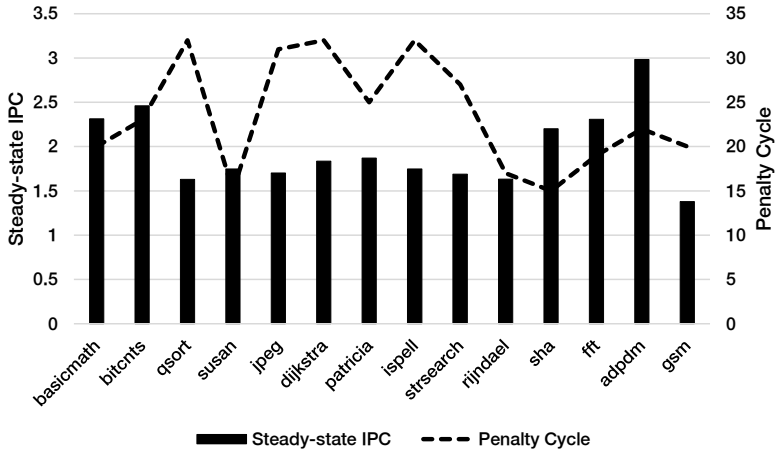


Figure 3.11: Steady-state IPC and Branch Misprediction Penalties for Benchmarks

approach.

Figure 3.11 shows that the steady-state IPC and the branch misprediction penalty significantly vary with each benchmark. For example, the steady-state IPC of *adpdm* is nearly double of that of *qsort*. Hence, estimating those values for each application is demanded.

To show how well the proposed technique predicts the performance difference between various configurations, we measured the prediction errors for the following four configuration changes: 1) the number of SimpleALUs is increased to 4, 2) the back-end pipeline width is reduced to 4, 3) instruction/data L1 cache size is decreased to 16KB, and 4) instruction/data L1 cache size is increased to 64KB.

The prediction error for four different configurations is presented in Figure 3.12, where the architecture A and B represent Cortex A15 and a new configuration respectively in the prediction error formula. According to the formula, the positive value indicates overestimation of IPC increase or underestimation of IPC decrease, whereas the negative value indicates underestimation of IPC increase or overestimation of IPC decrease.

It is observed that the average absolute prediction error is 1.9%, with a maximum of 11.1% (SimpleALU = 4). High prediction error observed with (SimpleALU = 4)

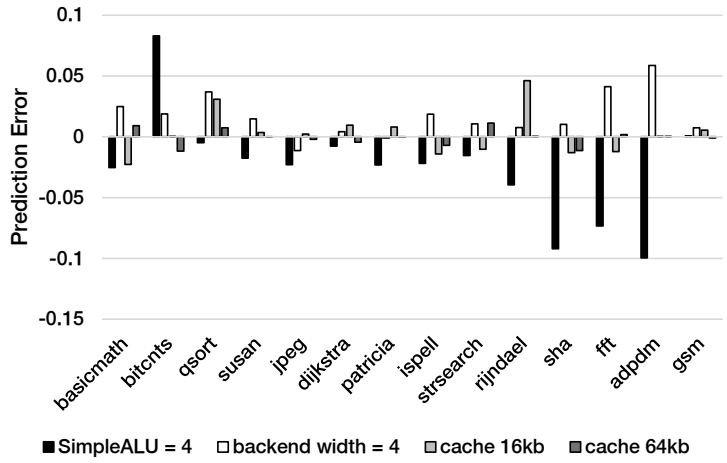


Figure 3.12: Prediction Error for Four Different Configurations

implies a discrepancy of the instruction issue mechanism of the trace analyzer and the reference simulator. Further improvement is required to ensure more reliable simulation results.

The main source of error is definitely the highly-abstracted processor model. In addition, some accuracy loss is inevitable if a functional emulator is used as the base simulator. For instance, as explained earlier, a functional simulator does not execute instructions on the mispredicted path, which may influence the cache simulator and branch predictor. Another cause of accuracy loss is discrepancies between the trace analyzer and the cycle accurate simulator. Timing accuracy would be improved if a more detailed mechanism of the target processor is modeled in the trace analyzer. For fair comparison with Gem5 syscall emulation mode that does not consider the system call overhead, we also ignored the system call overhead in the current implementation while it is possible to define a fixed system call overhead for each kind of system call.

For the reference, the simulation accuracy variations obtained from all sampling combinations are plotted in Fig 3.13.

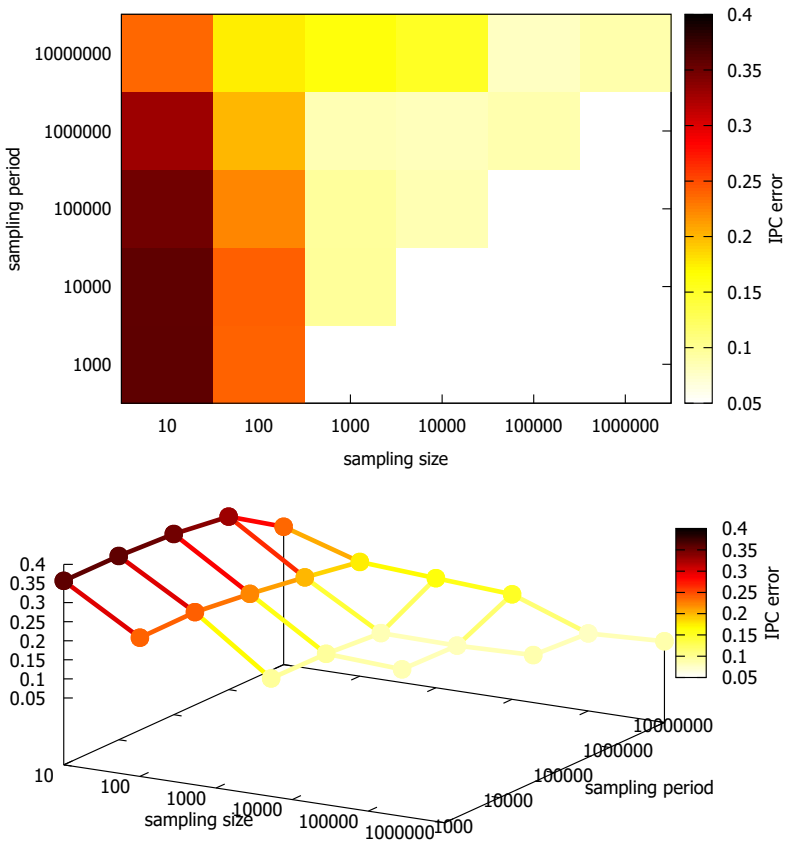
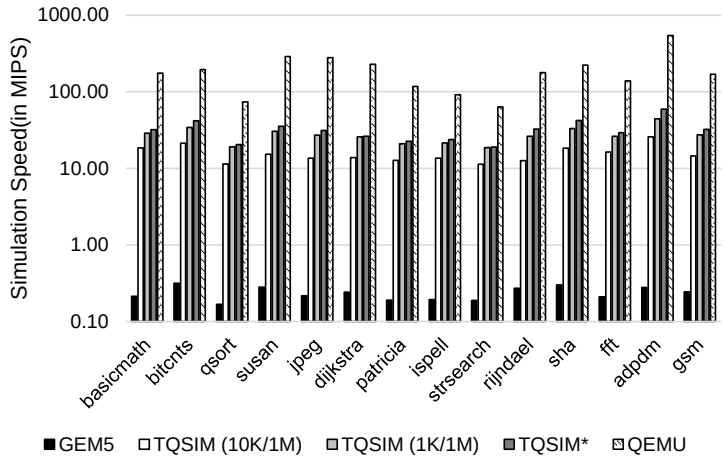


Figure 3.13: The Core Simulation Accuracy: All Sampling Combinations



	Gem5	TQSIM(10K/1M)	TQSIM(1K/1M)	TQSIM*	QEMU
Average	0.24	15.68	27.45	32.04	197.74
Minimum	0.17	11.36	18.62	18.99	63.74
Maximum	0.32	25.85	44.31	59.47	543.72

Figure 3.14: Core Simulation Speed (in MIPS)

3.5.3 Simulation Performance

We measured the simulation speed of TQSIM, along with Gem5 and the original QEMU. The simulation speed of TQSIM without the trace analyzer (TQSIM*) is also measured to examine the overhead of sampling and the trace analysis. The simulation speed is presented in Figure 3.14. The speed of TQSIM is 92 (10K/1M) - 159 times (1K/1M) faster than that of Gem5. Comparing the original QEMU and the TQSIM without the trace analyzer, we observed that instrumenting and calling helper functions to use cache/branch predictor slows down simulation by 3.36 - 9.14 times. Executing the trace analyzer with sampled instructions slows down simulation further 1.02 -1.34 times more. As we saw in the previous experiments, TQSIM demonstrates reasonable accuracy (around 8% error), while showing one or two orders of magnitude faster than a cycle-accurate simulator. This result confirms the usefulness of the proposed technique and TQSIM.

Board	ODROID-XU4	
Processor	Samsung Exynos5422 (ARM Cortex-A15 2.0GHz/Cortex-A7)	
L1 cache	Cache Size	32KB
	Set number	256
	Associativity:	2
	Block size	64
	Latency	1ns
L2 cache	Cache Size	2MB
	Set number	2048
	Associativity	16
	Block size	64
	Latency	30ns
Memory	2Gbyte LPDDR3 RAM PoP (750Mhz, 12GB/s memory bandwidth)	
	Latency	250ns

Figure 3.15: Reference HW Board

3.6 Discussion

We proposed a fast timed-simulation technique supporting modern superscalar out-of-order processors. For timing estimation, we use a novel combined analytical/sampled method that computes the simulation cycles analytically by using the steady-state IPC, which is obtained by the scheduling analysis of sampled traces. A functional simulator is extended with a cache simulator, a branch predictor, and the trace analyzer with which we estimate the dynamically varying steady-state IPC metrics. Experimental results with MiBench benchmarks prove that the proposed simulator TQSIM shows about 11.36 to 44.31 MIPS performance (up to 160 times faster than a cycle-accurate Gem5 simulator) with an average absolute error of approximately 8%. TQSIM is also capable of compromising between simulation time and accuracy by adjusting the sampling period and size. TQSIM is an open-source project currently available online [2].

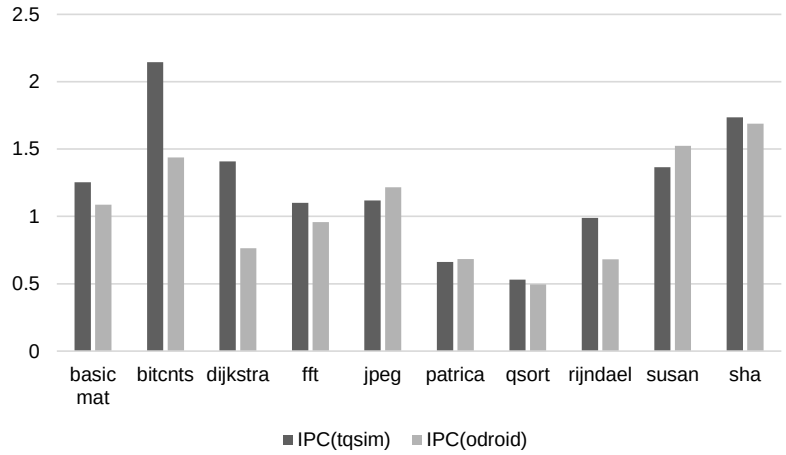
We aware that comparing the results of the presented approach to Gem5 may lead to more simulation error when the proposed simulator is compared to the real hardware. As a preliminary experiment, we evaluated the accuracy of the core simulator against the reference hardware board, which is described in Figure. 3.15

[2] As of December 2016, download at: https://github.com/cap-lab/tqsim

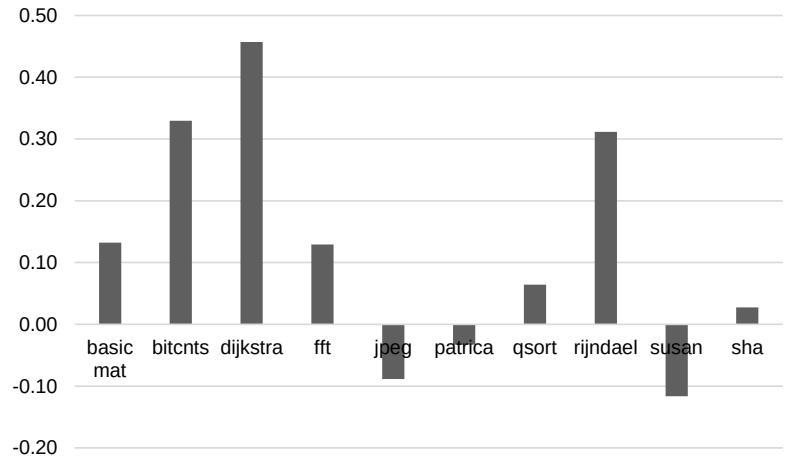
Simulated cycles from the hardware board and the core simulator are compared to validate the accuracy of the core simulator. To this end, we configured the board and the simulator to be the same configuration in Figure. 3.15. Most of latency parameters in the figure are obtained experimentally, implying that they would be inaccurate for some amount. For the benchmarks, we use Mibench suites. All printf() functions are disabled to minimize the error caused from I/O operations. To measure the simulated cycles from the board, we observe PMU registers. The experimental results are shown in Figure. 3.16.

The arithmetic mean of the absolute values of the IPC error is 16.9%, and the maximum value of the IPC error is 45.7% of *dijkstra* application. The arithmetic mean of the absolute values of the IPC error becomes 8.45% if we exclude three applications with a large error: *dijkstra*, *bitcnts* and *rinjdael*.

This experiment is a black box experiment, because it is hard to obtain any parameters and statistics from the board. If there had been a simulation board dedicated to simulation and debugging, the simulation accuracy could have been improved more precisely. We leave it as a future work.



(a) IPC Comparision



(b) Simulation Error

Figure 3.16: IPC Comparision and Simulation Error Obtained from the Core Simulator and Actual Hardware Board

Chapter 4

NoC Simulation

4.1 Network-on-chip

We briefly explain the basic concepts of Network-on-chip, before we provide the motivation, related works, and implementation details of the proposed Network-on-chip(NoC) simulator.

General NoC is generally composed of routers, which are interconnected by communication links that fulfill the communication. Usually a single NoC tile contains multiple IP cores, a single network interface, and a single router. Since each IP core may have a distinct interface protocol with respect to the network, a network interface is responsible for the logic connection between the IP cores and the network.

A packet is a message that a sender node wants to deliver to a receiver node. A packet is composed of a packet header and payloads to be delivered. The basic transmission unit of NoC is flits (flow control digits), so each packet is broken into multiple flits. The first flit, also called the header flit, carries information about this packet's destination and establishes the routing behavior for all subsequent flits. On the other hand, the final flit, also called the tail flit, closes the connection between the two nodes.

The most important module of NoC simulation is a router, which also consumes

the most of NoC simulation time. The number of ports contained in each router in the general mesh-type NoC structure are five: East, West, South, North, and Local. The first four ports are linked to other routers, while the local port is linked to a IP core, which produces or consumes messages.

Here, the operation of a router is explained based on the wormhole routing. A receiving process examines the links in each direction, identifies the new flit, and copies the flit if the buffer is not full. A sending process is divided into two parts: reservation and forwarding. Reservation literally reserve the path (from input port to output port) of packet in the router. When the first flit of the packet is entered, the router tries to reserve the path/route based on the packet header included in the first flit. If the number of possible routes is multiple, the single route is chosen using the specified selection algorithm. Forwarding procedure forwards following flits in the input buffer to the output port based on the reserved route. If the last flit of the packet is left, the reserved route from input to output is released. In this way, flits, which do not have a packet header, are routed in the correct direction.

One of the biggest weaknesses of the worm-hole routing is that while a route from an input port to an output port are reserved, other packets cannot use the output port at all. To mitigate this problem, the notion of virtual channel was invented and widely used [45]. The basic idea of the virtual channel is that the physical channel of each direction is multiplexed into multiple virtual channels to be shared on flit-by-flit basis. This scheme enables better network latency and throughput.

4.2 Motivation

For a large-scale many-core architecture, an efficient connectivity is a major challenge. Previous interconnects including buses, all-to-all point-to-point connections, and even rings, clearly does not scale beyond a few nodes. Network-on-chip(NoC) architecture brings notable improvements over conventional bus and crossbar interconnections especially for many-core architecture, which integrates a high number of components. NoC improves the scalability, power efficiency, and support to globally asynchronous

locally synchronous (GALS) paradigm. Practically, most of many-core architecture simulators that we have reviewed support flit-level or packet-level NoC simulation at least.

For many-core architecture design, we can no longer ignore communications inside the memory hierarchy and the interconnection network. If deliberately ignoring these interconnect details, the reliability of the many-core architecture simulation is greatly compromised. The experimental results in [7] show that using simple network model generates an inaccurate total simulated cycle, eventually leading to a wrong system design choice. For this reason, a detailed and accurate interconnection network model within a full-system evaluation framework is essential.

Since most communication between cores occurs through NoC, the speed of NoC simulations has a critical effect on the speed of the entire many-core simulation. However, the high-performance NoC simulation is a very challenging task due to the analogous reasons discussed with many-core simulation. The biggest reason is that there are too many concurrent components to be simulated every cycle. Although it may be possible to try multi-threaded implementation, the performance gain obtained from such parallelization is disappointing due to limitations which can be explained later in the thesis.

For accurate NoC simulation, it is essential that the accuracy of the packet latency must be guaranteed. The most important factor determining the latency of each packet is the congestion status of the network. In a computer system, if the total sum of demands on a resource is more than its available capacity, the resource is said to be congested. There are two main types of congestion on NoC: link congestion and buffer congestion. Link congestion occurs when two or more packets compete with the same channel, while buffer congestion happens only when flits contend for the use of limited buffer. Packet-level simulation is sufficient to model link-congestion, but flit-level simulation is required to model buffer congestion. It is intuitive that flit-level simulation is much more slow than packet-level simulation. For your information, Sniper [12] is one of representative many-core simulator which supports only packet-level

Table 4.1: Comparison of State-of-the-arts Many-core Simulators (revisited)

	Scope	Timing Model	Parallelism	Network Model	Speed
Gem5[4]	User/Full	CA	Seq	Flit-level	<< 1 MIPS
Marss[8]	Full	Hybrid	Seq	Flit-level	<< 1 MIPS
Sniper[12]	Full	DBT+Instr.	Loose	Packet-level	2 MIPS
Zsim[17]	User	DBT+Instr.	Loose	Fixed-delay	41 (avg), 300 (max) MIPS
Manifold[18]	Full	Any	Cnsv/Loose	Flit-level	<< 1 MIPS
Hornet[22]	User	CA	Cnsv/Loose	Flit-level	Not available
Ours	User	DBT+Instr.	Cnsv	Flit-level	6 (avg), 32 (max) MIPS

* Abbreviations) CA: Cycle-accurate / Seq: Sequential / Anlt: Analytical / Samp: Sampled
Cnsv: Conservative, DBT+Instr: Dynamic binary translation + instrumentation

NoC simulation.

We observed, through surveys and simple laboratory experiments, that the most available NoC-dedicated simulator shows less than 100KCycles/s performance even with very low packet injection rate such as 0.000391. As a result, most of the available many-core simulators supporting flit-level NoC simulation have hundreds of KIPS performance.

Based on this motivation, we developed and evaluated various implementations and optimization techniques to speed up NoC simulations without losing the accuracy of the simulation.

4.3 Related Works

In this section, we review state-of-the-arts open-source standalone NoC simulators: Noxim [46][47], Booksim2 [48], and Garnet [7]. Specifically, we present how actual NoC simulation is performed at the level of code. In addition, code availability is an important feature especially for research purposes.

The three simulators support all the most important elements to be equipped with NoC simulator:

- Various topologies are supported and can be easily included.
- Various parameters are configurable. They support buffer size, network size,

packet distribution, packet injection rate, packet size, routing algorithm, selection strategy, traffic distribution

- Statistics including latency, throughput, and (optionally) power-consumption are available.

4.3.1 Noxim

Noxim [46][47] is a network-on-chip simulator implemented in SystemC. The choice of SystemC was motivated by some of its key features, such as the strong and widely tested simulation scheduler, the support for various abstraction levels including low-level RTL design, the intrinsic parallelizability of the execution model, and the Transaction-Level Modeling (TLM) capabilities.

Initial version of Noxim supported only mesh architecture, which restricts the analysis of other topologies. However, recent Noxim has been modified to support various other topologies and to employ different routing algorithms. According to [47], Noxim has been employed as an experimental platform, or as a part of an experimental infrastructure, in more than 400 scientific papers published in refereed ACM and IEEE international conferences and journals.

The main function of each router is implemented as the following code:

```
1  SC_CTOR(NoximRouter) {
2      SC_METHOD(rxProcess);
3      sensitive << reset;
4      sensitive << clock.pos();
5
6      SC_METHOD(txProcess);
7      sensitive << reset;
8      sensitive << clock.pos();
9
10     SC_METHOD(bufferMonitor);
11     sensitive << reset;
12     sensitive << clock.pos();
13 }
```

It is a structure that calls three processes per each positive clock edge. Executing rxProcess, txProcess, and bufferMonitor processes, SystemC scheduler traverses all routers, and then increases the simulated cycle by one.

4.3.2 Booksim2

Booksim2 [48] is a detailed, cycle-accurate simulator written in C++ language. The authors claimed that Booksim2 is one of the first works that validate the simulation results of a network simulator against an actual RTL implementation. The latency and throughput statistics generated from Booksim2 closely match those of the RTL model. Errors in latency measurements were less than 5%.

In the source code of Booksim2, the following step function is called every cycle.

```
1  _step () {
2      ....
3      for ( int subnet = 0; subnet < _subnets; ++subnet ) {
4          for ( int n = 0; n < _nodes; ++n ) {
5              .....
6              //work
7              ....
8          }
9      }
10     ....
11     for ( int subnet = 0; subnet < _subnets; ++subnet ) {
12         for ( int n = 0; n < _nodes; ++n ) {
13             .....
14             //work
15             ....
16         }
17     }
18 }
```

Traversing every node in every subnetwork, the simulator performs transmission and reception of flits. For this reason, the author admits that BookSim can be slower compared to some of the other simulators. The authors also claimed that network simulation is often several orders of magnitude faster compared to a multiprocessor system simulator or a full-system simulator and is not necessarily the bottleneck. However, many current many-core simulators, including Sniper, Zsim, and ours, support high-speed core simulation (tens of MIPS performance), and multi-threaded simulation. As a result, NoC simulation is becoming a bottleneck in full-system simulation.

4.3.3 Garnet

Garnet [7] is a detailed network simulator that is incorporated into the Gem5 full-system simulator [4] and is also available as a standalone network simulator.

Unlike Noxim or Booksim, which has a code block that is executed every cycle, Garnet uses a event-driven model to perform timing simulation. Since various components communicate using message buffers, the component at the receiving end of the buffer can be scheduled to wake up when the message is available to be read from the buffer. Therefore, the network simulation proceeds by registering and scheduling events on the event queue.

Therefore, Garnet is expected to operate at a higher speed than Noxim or Booksim for low to medium network load. However, as the packet injection rate increases, the gain disappears and the cost for managing the timed queue may increase.

4.4 Proposed Approach

4.4.1 Implementations

In this section, we introduce various structures of NoC simulators: kernel-level thread implementation, user-level thread implementation, and sequential implementation. In addition, we explain the technical difficulties in a fast NoC simulation.

Kernel-level Thread Implementation

The naive way to implement NoC simulator based on kernel-level threading is to allocate a network interface/router in a tile to one kernel-thread and to perform a lock-step synchronization. The operating system on the host machine manages each kernel-thread, so it is possible to utilize multiprocessor systems by splitting threads on different processors or cores.

We assume that a network interface and a router receive a flit at the rising edge of the clock signal, and send a flit at the falling edge. For clock synchronization, we located a barrier at each clock edge. Therefore, all kernel threads are synchronized twice at each cycle.

The sample pseudo code for each thread is given as follows:

```
1  while (true){
2
3    // n: a network interface, r: a router
4
5    n.rxProcess()
6    r.rxProcess()
7
8    barrier()
9
10   r.txProcess()
11   n.txProcess()
12
13   barrier()
14 }
```

Preliminary experiment shows that the simulation speed maintains around 30KCycles/s, regardless of the packet injection rate. This result confirms that NoC simulation cannot be implemented with a naive kernel-thread parallelization for three reasons: 1) the amount of computation each thread does between barriers is too small, 2) two synchronizations per each cycle incur large time overhead, and 3) kernel-level threads are heavier than user-level threads because of expensive kernel-level context switch.

User-level Thread Implementation

We observed that the implementation based on kernel-thread is not applicable due to the management overhead in the last subsection.

Just as kernel threads are managed by the OS kernel, user-level threads are managed by the user-level application. User-level threading runs at a very high speed by allowing no context-switching and no kernel intervention. However, since it is a single-threaded process from the OS kernel point of view, you cannot use multiple host cores as you would with a kernel-level thread. Quickthread incorporated in SystemC framework and protothreads [49] are representatives of libraries supporting such user-level threading.

To test feasibility of user-level threading, we performed a preliminary experiment using SystemC library. We designed very simple thread that increments a local variable every cycle. We measured the simulation speed as the number of threads

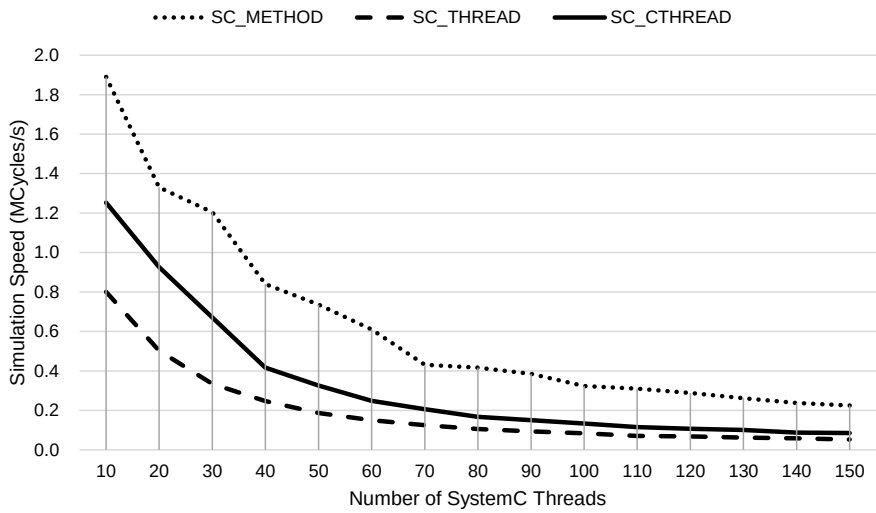


Figure 4.1: Simulation Performance as the Number of User-Level Threads. SystemC Library Supports Two Types of Threads: SC_THREAD and SC_CTHREAD

increases. As SystemC library supports two types of threads, SC_THREAD and SC_CTHREAD, we tested both types. While SC_THREAD is activated by any types of events, SC_CTHREAD is activated by only specified clock edge. The results is presented in Figure 4.1.

It is observed that the simulation cannot perform faster than 300KCycles/s once the number of threads exceeds 60. In the case of an 8x8 NoC architecture, the number of SystemC modules is at least 128, including 64 router modules and 64 IP core modules, which are attached to the routers. Given the fact that this experiment assumes that the computation workload of each thread is minimum, the NoC simulation with actual router/network implementation will be much slower.

Sequential Implementation

As explained, the small amount of computation each thread does between barriers hinder parallel simulations of NoC. Therefore, we implemented the NoC simulator sequentially without any explicit threads. As mentioned earlier, Booksim is implemented in this form.

The main loop of the sequential NoC simulator is given as follows:

```
1  while (true){
2    foreach n in NIs: // for all network interfaces
3      n.rxProcess()
4    foreach r in routers:
5      r.rxProcess()
6
7    foreach r in routers:
8      r.txProcess()
9    foreach n in NIs:
10     n.txProcess()
11   cur_cycle++
12   if (termination_condition)
13     break
14 }
```

For sequential implementation, the order in which you call each function is very important. If the order is not carefully determined, something that cannot happen in a cycle can happen in a cycle. For example, unlike the presented code, if a txProcess is called first and then a rxProcess is called, sending and receiving of a single flit can occur simultaneously in a cycle. This flow is like the one of the main-loop of the out-of-order processor simulator, which executes the write-back pipeline first and executes the fetch pipeline at the end, in the reverse order of the actual pipeline.

As the number of concurrent components increases in the target NoC, the simulation time to proceed one target cycle also increases. It is a reasonable assumption that it takes 50 host cycles to simulate a router or a network interface just for one cycle. It means that the host computer requires us 6,400 cycles to simulate 1 cycle of 8x8 NoC. Since 6,400 cycles takes 6.4 us with a host machine with 1GHz clock speed, the NoC simulation speed is just 156,250 cycles/s with 1GHz host machine. Therefore, performance optimization for NoC simulator is absolutely necessary.

4.4.2 Optimizations

Here we present two NoC simulation optimizations techniques. Performance enhancements for each technique are presented in the evaluation section

Event-driven Process Scheduling

In the sequential main loop of the NoC simulator, all routers and network interfaces schedule receiving/sending process at each cycle. We observed that simulating every concurrent component in a NoC architecture at each cycle is very expensive. When the packet injection rate is very low, most routers and network interfaces remain idle. Hence, it is able to schedule processes that are certainly active in a next cycle, skipping processes that are certainly idle in a next cycle.

In this way, if active and idle components are separated and managed centrally, we can skip simulating idle routers and network interfaces in the main simulation loop. This simple optimization significantly reduces the simulation time for each loop.

To simplify the explanation, we will briefly explain the idea through the router that takes the most simulation time in NoC simulation.

Once a packet header reaches a router, it performs route reservation with this information. If a path from the west port to the north port is reserved at this time, it means that the router in the north direction will receive flits within one or several cycles. Therefore, we call rxProcess of the router in the north direction within the next cycle. This idea is the same as the basic idea of queue-based scheduling in Garnet.

On the other hand, routers are generally not the destination of packets or flits. Therefore, if the central simulation manager monitors the number of flits in buffers of each router, the manager can determine which routers will send a packet within one or several cycles. Therefore, we call txProcess of such routers within the next cycle.

This technique can be used regardless of specific routing techniques.

Object (Packet/Flit) Pool Pattern

Packet/flit pool optimization borrows a concept of the object pool pattern from the software creational design patterns.

Once a component has a message for another component, the connected network interface generates a packet and a series of flits to be transmitted. The memory space for the packet/flits are dynamically allocated at the source tile, and deallocated at the

target tile. Single memory allocation overhead is negligibly short. However, in case that the speed is paramount, frequent dynamic allocations and deallocations become very expensive. Moreover, a creation and destruction of a packet/flit is repeated in a very short amount of time.

Object pooling can offer a significant performance in situations where 1) the cost of initializing a class instance is expensive and 2) the rate of instantiation and destruction of a class is high. Each reuse saves a significant amount of computation time and memory spaces.

Therefore, we let the NoC simulator maintain a set of initialized packets/flits kept ready to use a pool, rather than allocating and destroying them on demand. A client of the pool, a network interface, will request an object from the pool. If a previously prepared object is available it is returned immediately, avoiding the instantiation cost. If no objects are present in the pool, a new item is created and returned. The network interface fills the packet header and payloads on the returned object. When the transmission has finished, the network interface at the destination tile returns the object to the pool rather than destroying it.

4.5 Experimental Results

4.5.1 Impact of Implementations and Optimizations

In Figure 4.2, NoC simulation times to simulate 1 million cycles are presented according to various implementation types and optimizations. X axis denotes the packet injection rate, which indicates how many packets are generated every cycle at each IP core. With the lowest packet injection rate 0.000391 in the figure, the total number of simulated NoC packets becomes around 25098, and with the highest packet injection rate 0.0125, the total number of packets becomes around 799,452. Each packet contains 2 - 10 flits, which is randomly determined in these experiments.

Noxim is developed based on TLM 1.0 SystemC standard, scheduling with user-level threading. Once we redesigned it from user-level threading implementation to sequential implementation, the NoC simulation becomes 2.55 (packet injection rate

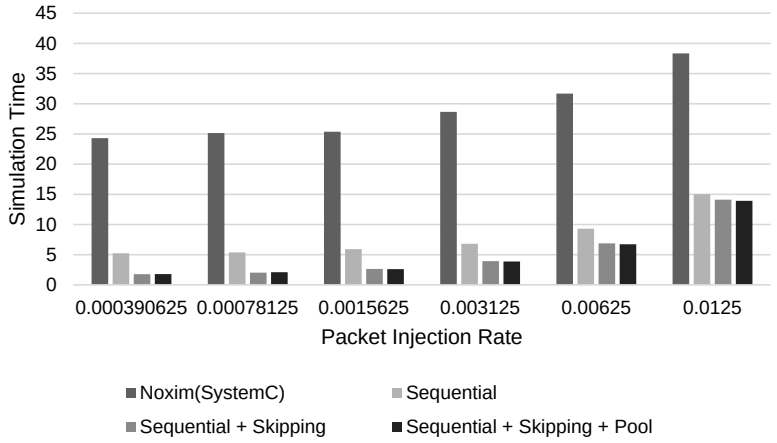


Figure 4.2: Impact of NoC Implementations and Optimizations

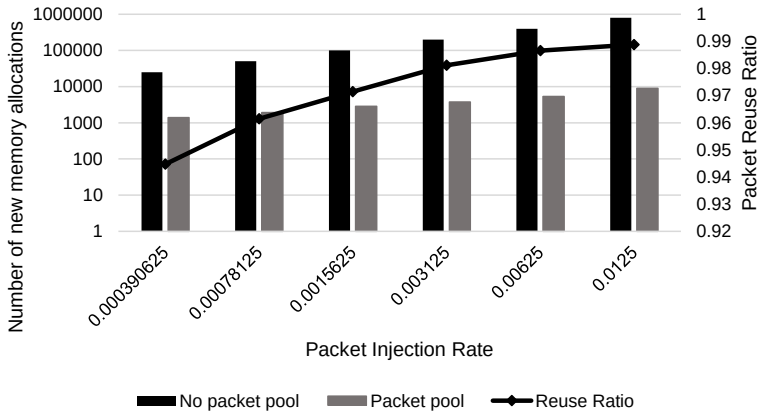


Figure 4.3: Numbers of Memory Allocations and Memory-Reuse Ratio For Packet Generations

= 0.0125) - 4.64 (packet injection rate = 0.000391) times faster. In addition, dynamic skipping and packet/flit pool optimization accelerates the simulation even more by 1.07 (packet injection rate = 0.0125) - 2.92 (packet injection rate = 0.000391) times. Skipping is more effective optimization technique than the memory pool, and all optimizations are more effective when the packet injection rate remains low.

However, the use of memory pool significantly reduces the number of new allocations. As shown in Figure 4.3, 94.5 % - 98.9 % of packets are recycled.

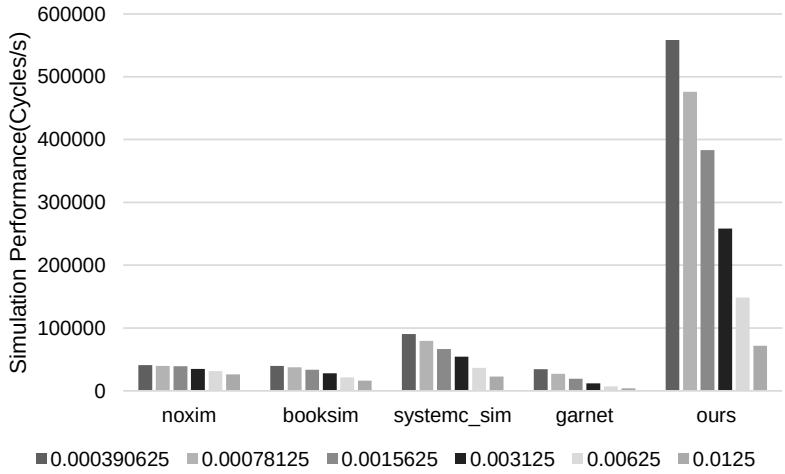


Figure 4.4: The Simulation Performance of the Proposed NoC Simulator and Various State-Of-The-Arts NoC Simulators

4.5.2 Comparison with Other State-Of-The-Arts

For the purpose of comparison, Noxim, booksim2, Garnet and one in-house simulator were tested on the same configuration. The in-house simulator is a user-level threading simulator based on SystemC, which is written in TLM 2.0 SystemC standard.

All NoC simulators, including ours, are configured to simulate 8x8 NoC topology. The accuracy of all simulators is considered to be same for all flit-level NoC simulators. Our NoC simulator is compared to the existing state-of-the-arts simulator in Figure 4.4. As a result, the proposed simulator performs 2.75 (compared to Noxim, packet injection rate = 0.0125) - 21.36 (compared to Garnet, packet injection rate = 0.003125) times faster. We also can notice that most available flit-level (dedicated) NoC simulators are slower than 100KCycles/s even with extremely low packet injection rate (0.0004 packets/cycle).

Of course, these comparisons are not entirely fair, because Noxim and Booksim support various topologies, routing/selection techniques, and more detailed statistics. In contrast, our simulator assumes that the NoC structure is only mesh, and the routing algorithm is XY routing.

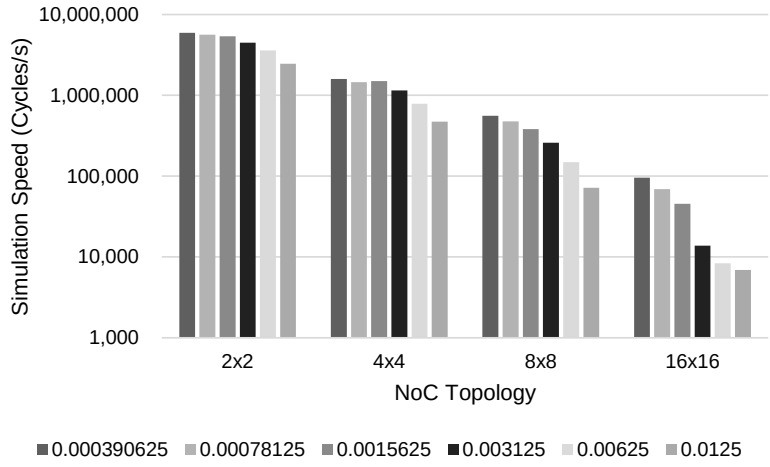


Figure 4.5: Proposed NoC Simulator's Performance For Various Topologies and Packet Injection Rates

4.5.3 Performance Evaluation For Various Configurations

The speed of the NoC simulation is determined by the complexity of the network configuration and the activity in the network. Therefore, we measured proposed NoC simulator's performance for various topology and packet injection rate.

Our NoC simulator works at 100KCycle/s - 500KCycle/s until the packet injection rate exceeds 0.01. We have implemented a high-performance flit-level NoC simulator that outperforms others. However, when the packet injection rate is over 0.01 or the NoC topology is bigger than 8x8, the performance of the proposed simulator is markedly declined.

4.5.4 Full-System Simulation Accuracy Impact

Some many-core simulators such as Zsim [17] perform a naive NoC simulation using fixed latency model. However, the accuracy validation of the simulator in this thesis is not done through many-core NoC architecture for practical reasons. The fixed latency NoC model does not have any consideration for the contention that may occur during network simulation. Therefore, we designed an experiment to confirm the accuracy impact of using a fixed latency model.

Table 4.2: Full-system Simulation Error with Fixed Latency Model

	L1 data cache miss rate	Packet injection rate	Error
capp1	0.03	0.023433	36%
capp2	0.0002	0.008	31%
ncapp		0.0005	0%

For the fixed latency model, the zero-load latency of every possible source and destination combinations is measured to make the default latency between the nodes. Therefore, when a packet is generated during full-system simulation, the packet arrives at the destination node after this zero-load delay.

In this target architecture, NoC communication is mainly caused by memory accesses. Thus, we have created three memory-intensive synthetic examples. All benchmarks perform continuous memory copying. Both *capp1* and *capp2* access cachable memory addresses, while *ncapp* accesses noncachable memory addresses bypassing level 1 and 2 data caches. For *capp1* benchmark, the size of data cache is minimized to maximize L1 cache misses; for *capp2*, the size of data cache is increased to avoid misses. The experimental results are given in Table 4.2.

In the examples of *capp1* and *capp2*, at least 30 % of simulation error is observed. Note that the target architecture does not support hardware-level cache coherency. Cache coherent architecture generates frequent cache invalidations, increasing injection rate and simulation error even more. The Garnet paper [7] reports that the simulated cycle difference between a simple network model and a detailed network model is up to 4 times.

In *ncapp* example, the memory read command makes the to wait for the respond, so the injection rate remains very low, resulting in small packet latency variations. Even though a packet arrives later in memory because of the increased packet latency, it will eventually become serialized in memory because it must wait for memory to process past accesses because of the limited number of concurrent memory accesses. The memory access time (about 200 cycles) itself is so huge that the impact of longer

packet latency is also reduced.

On the other hand, timing variations in the network can cause parallel applications to take different execution paths, causing differences in the number of instructions and memory accesses [7]. For example, the different order in which threads attain spin locks creates completely different simulation results.

Therefore, an accurate network model is very important to capture realistic and reliable timing characteristics inside a full-system simulation.

4.5.5 Accuracy

The proposed NoC simulator is a cycle-accurate simulator. Thus, since there is no approximation introduced by the system model, the estimation of performance figures (e.g., latency and throughput) is error free.

In addition, we observed that the NoC model included in the proposed simulator presents identical statistics generated from Noxim simulator. This is because our NoC simulator was written on the basis of Noxim code.

4.6 Discussion

Implementation of fast NoC simulator is still a challenging task. Most available flit-level many-core simulators, including Gem5, Marss, and Manifold, shows only hundreds of KIPS performance. Other many-core simulators resorted to using fixed-delay model or analytical model. We agree that the performance of our NoC simulator is still dissatisfying for some cases. However, it is predicted that a faster flit-level simulation than the presented one is extremely difficult.

There exist two orthogonal ways to reduce the workload of the NoC simulation.

First, we can use an analytical model [50], [51] or empirical model[52] to estimate the packet latency. In this case, the accuracy of the entire simulation depends on the accuracy of the model. One of the most naive models will be a hop-based static latency model. In this case, the packet latency is obtained by (the number of hops between the source and destination tiles multiplied by the minimum packet latency for a single hop).

When live packets within the network is very sparse, the accuracy may be reliable, but once NoC gets congested, the accuracy of the packet becomes very poor, as we have shown in the experiment.

Second, we can perform abstracted NoC simulation for the non-interested regions. When performing simulations, all parts of the application are not equally important from a developer's point of view. Based on this case, it is possible to perform a detailed simulation only on the region of interest (ROI), and perform a abstracted simulation otherwise, which can speed up the simulation of the entire application. This technique is known as a fast-forwarding, and it is widely adopted in many state-of-the-arts simulators including Marss, Gem5, and Zsim. In our case, it is possible to perform flit-level detailed NoC simulation for ROIs and a hop-based static latency simulation outside of ROIs.

Chapter 5

Simulation Backplane

5.1 Overview

Generally, embedded software developers use the instruction-set simulator (ISS) and cross-compiler dedicated for the target instruction-set-architecture (ISA) to develop software. On the other hand, embedded hardware designers develop hardware components using the hardware description language(HDL) simulator. Since an instruction-set simulator is lack of micro-architectural details including cache, interconnect, and memory architectures, the ISS is not intended to use for performance evaluation. In order to solve this problem, ISSs and HDL simulators are combined into a single HW/SW co-simulation platform, which enables HW/SW co-development and early-stage design space exploration. Although the HDL simulator has the advantage of expressing a concurrent characteristic of HW components, it is too hard to describe every detail of a hardware component, and the speed of the HDL simulator is too slow.

SystemC is an open source C++ HDL. Two of the biggest advantages of the SystemC environment are that it enables developers to write a HW Module in C++ language that is familiar with the software developer, and it supports various levels of abstraction from transaction level modeling (TLM) to register transfer level (RTL) de-

sign flow. Utilizing the advantages of the SystemC, the ISS-SystemC co-simulation framework is widely used for rapid development and simulation.

In this study, the target architecture is virtually prototyped with the ISS-SystemC co-simulation framework. The simple overview is given in Figure 5.1.

The interprocess communication and the core wrapper are two key enablers to ISS-SystemC co-simulation framework. Interprocess communication realizes the communication between an ISS and the SystemC simulator, that run as distinct processes on the host system. On the other hand, the core wrapper ensures synchronization between the SystemC backplane and a component simulator, and translates the information coming from ISS into cycle-accurate timed event.

There exist two serious problems to hinder efficient co-simulation: 1) heavy intercommunication overheads between the two heterogeneous simulators would significantly slow down simulation performance, 2) SystemC scheduling overhead become a serious bottleneck as the number of HW components increases.

In this dissertation, we use trace-driven co-simulation, the shared memory IPC, and virtually segmented L1 cache to reduce the heavy communication overhead. To be more specific, trace-driven co-simulation / virtually segmented L1 cache reduce the number of intercommunications, and the use of shared memory IPC speeds up an individual intercommunication. To solve the second problem, we present event-based slave module modeling. Then, we also study the feasibility of SystemC parallelization

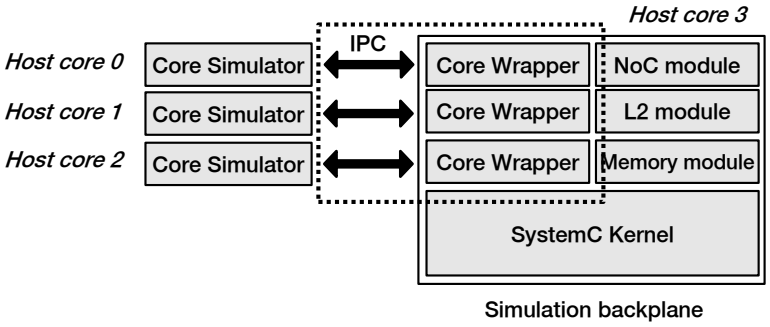


Figure 5.1: The Proposed ISS-SystemC Cosimulation System

techniques with timing decoupling.

In this chapter, we describe the brief description of SystemC, which is a description language for the simulation backplane. We then discuss how to configure simulation backplane with SystemC. We also present which types of optimization have been performed to enhance the performance of simulation backplane.

5.2 Background

5.2.1 SystemC

When modeling the system in SystemC, the function is modeled as a set of processes and the structure is modeled as a set of modules. The processes are defined in the module, and they are communicated by means of a port/channel (RTL abstraction level) or a socket/callback function (TLM abstraction level). These processes are simulated as they are performed concurrently.

SystemC modules (sc_module) are the basic building blocks that compose a system, and internal behavior of each module is modeled by processes. Processes are classified into two categories: SC_METHOD, and SC_THREAD[1]. The former performs instantaneous computations, and it cannot be suspended and resumed in the middle of execution. On the other hand, a SC_THREAD process can be suspended by using *wait* SystemC API until the specified time elapses or the specified event takes place. It behaves like a software thread.

The connection between modules determines the level of abstraction: RTL or TLM level. At the RTL level, the interaction between modules are modeled by a pin-level description by using a port and signal; At the TLM level, the interaction between modules are modeled by a function call. I would like to explain only the latest TLM 2.0 standard in this thesis.

[1] SC_CTHREAD can be regarded as a special case of SC_THREAD, which is sensitive to a clock.

5.2.2 OSCI Transaction Level Modeling Standard 2.0

Since TLM-1.0 standard suffers from the lack of standard for modeling transactions and coding styles, different implementations are not compatible with each other [3]. The OSCI TLM 2.0 standard addresses several of the shortcomings of the TLM 1.0 standard with respect to model interoperability and simulation speed. OSCI Transaction Level Modeling standard 2.0 (TLM-2.0) introduces two important concepts: socket and generic payload.

Two modules exchange transactions through sockets. Once a socket of an initiator is bound to a socket of a target module, the initiator is able to send out transactions through the initiator socket, and the target receives incoming transactions through the target socket. At the low level, the initiator module makes a function call with several arguments such as transaction payload, current transaction phase, and delay. The callback function registered to the target socket is invoked to serve the transaction. The generic payload is introduced to improve the interoperability of memory mapped bus models by defining the standard transaction objects passed through the sockets.

Function calls to a socket can be categorized into two types: *blocking* and *non-blocking* transports. If the initiator module calls a blocking transport function, the initiator becomes blocked until the target module returns the control after processing the transaction. If the initiator module calls a nonblocking transport function, the target module registers the payload of the transaction with the processing time to the payload event queue. After the processing time elapses, the transaction is processed by the callback function registered with the payload event queue, and the target module calls a non-blocking transport function on the backward path to inform the initiator module of the completion of the transaction.

Two coding styles are documented in the reference manual of TLM-2.0 [3]: *Loosely Timed* (TLM-LT) and *Approximately Timed* (TLM-AT). By using blocking transports for communication, TLM-LT supports *temporal decoupling* that allows a SystemC process to run ahead of the global clock, which can improve the simulation speed. This

coding style sacrifices timing accuracy so that it is mainly used for software development. On the other hand, TLM-AT uses non-blocking transports for more accurate timing model, which is necessary for architectural exploration and performance analysis.

5.2.3 Synchronization Techniques

A parallel discrete event simulation can be viewed as a set of sequential discrete event simulations that exchange time-stamped events. Synchronization problem is to devise an algorithm to ensure each simulator processes events in time stamp order, without violating the causality constraint. If events that have causal relationship are processed out of timestamp order, the causality constraint is broken.

The synchronization technique is critical to the performance and accuracy of the parallel simulation. Extensive researches have been performed to reduce the synchronization overhead and increase parallelism.

The representative synchronization techniques are given as follows:

Lock-step

The lock-step method synchronizes all component simulators, which are executed in parallel, every global cycle. It is the simplest synchronization technique and is used in many commercial parallel simulators as it guarantees the most accurate results without causing causality errors. Since each component should be synchronized every cycle, the computational resource allocated to each component simulator is significantly wasted and the frequent interprocess communications are performed. Therefore the performance is extremely poor and there is little or often negative gain from parallel simulation.

Conservative

The conservative approach [53]–[56] refers to a synchronization method that guarantees that no past event will occur at present. To do this, the local clock of each component simulator can proceed up to the earliest timestamp of all possible events in the

future. However, it is practically impossible to know all the events that may occur in the future. Therefore, the efficiency and performance of the conservative method depends on how the predicted minimum time, called lookahead time, is calculated.

optimistic

The optimistic approach [57]–[59] starts with the assumption that past events will rarely arrive at present. Thus, each component simulator will advance its local clock until it is obvious that a causality error has occurred. As a result, each component simulator has better parallelism by reducing the number of synchronization. Then, when a causality error occurs, all component simulators roll back to the last checkpoint. However, checkpointing and rollback mechanism of the component simulator is hard to implement. It is extremely hard to rollback to the consistent state of the system. Moreover, simulation performance with optimistic approach is deteriorated when causality error occurs frequently.

Loose

We define a new category "loose synchronization" which does not strictly enforce the ordering of all events in the system. For example, quantum-based (or barrier-synchronization) technique synchronizes every quantum (or at every barrier), while slack simulation technique [60] forces all simulators to stay within a cycle window whose size is the maximum slack. In point-to-point synchronization scheme proposed in [61], each simulator periodically chooses another simulator at random and synchronizes with it. If the clocks of the two simulators differ by more than a configurable number of cycles (called the slack of simulation), then the simulators that is ahead goes to sleep for a short period. Although loose synchronization approaches sacrifice the accuracy as trade-off, various many-core simulators usually use this loose synchronization thanks to its high-speed.

Most of literatures, which are cited here, are extensively discussed in [62].

5.3 SystemC Models for the Target Architecture

The target architecture and the proposed SystemC model for the target architecture is presented in Figure 5.2 and 5.3, respectively.

A core wrapper module corresponds to each core simulator. Core wrapper module is responsible for communicating with a core simulator, and transferring the request

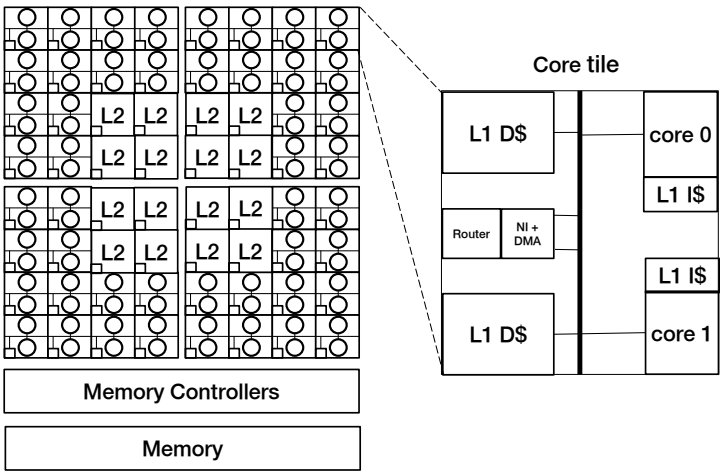


Figure 5.2: Simulated Target System Characteristics (Revisited)

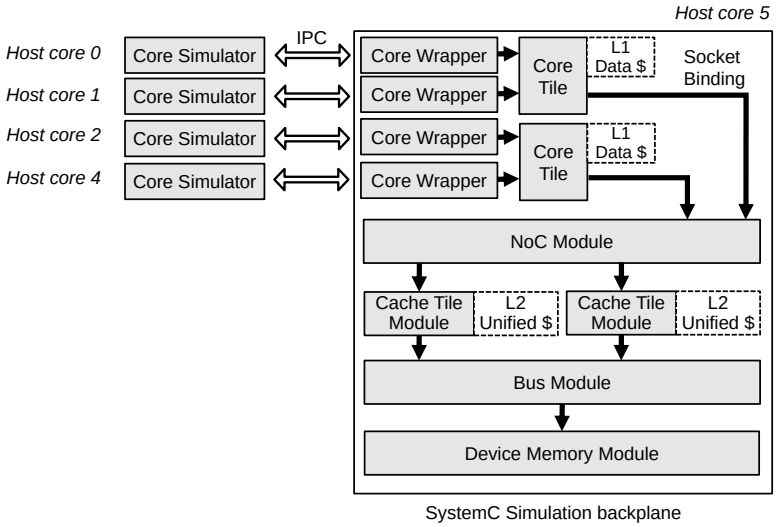


Figure 5.3: SystemC Models For Simulated Target System

to the backplane in the chronological order and time. A core tile module or cache tile module are defined depending on the type of tile modules. Core tile module is responsible for processing L1 cache operation and generating a set of L2 cache requests if cache miss, flush or write-back event occurs. Similarly, cache tile module is responsible for processing L2 cache operation and generating a set of memory requests if cache miss, flush or write-back event occurs. Bus module is the interconnection module to connect L2 caches, arranges requests from multiple L2 caches, and transfers them to the device memory module. The device memory module manages the device memory. NoC module contains a single process NoC simulator, which will be described in the next section. NoC module generates and processes NoC packets once a core or (a) cache tile sends a request or response transaction.

Each module is connected via sockets according to TLM-2.0 standard, and communicates with nonblocking function call based on approximately timed model. If module A sends a request to module B, the transaction with a timing annotation is registered to the payload event queue in module B. Once the time-stamp of the transaction becomes the simulated time, the module B serves the transaction.

The total number of modules in the system is 96(core wrapper) + 48(core module) + 16(cache module) + 1 (NoC module) + 1 (bus module) + 1 (device memory module) = 163.

However, as the number of modules increases, the speed of SystemC simulation, which is basically performed in a single process, decreases significantly. The baseline SystemC simulation kernel provided by the Open SystemC Initiative (OSCI) [63] is to use a system-wise global event queue and servicing the events sequentially.

Figure 5.4 presents the simulation performance of the OSCI implementation with a simple benchmark that contains independent counters. It is observed that the simulation speed decreases to below 200 KCycles/s as the number of components increases over 50, even though the computation time of each module is minimal in the benchmark and no inter-module communication is involved.

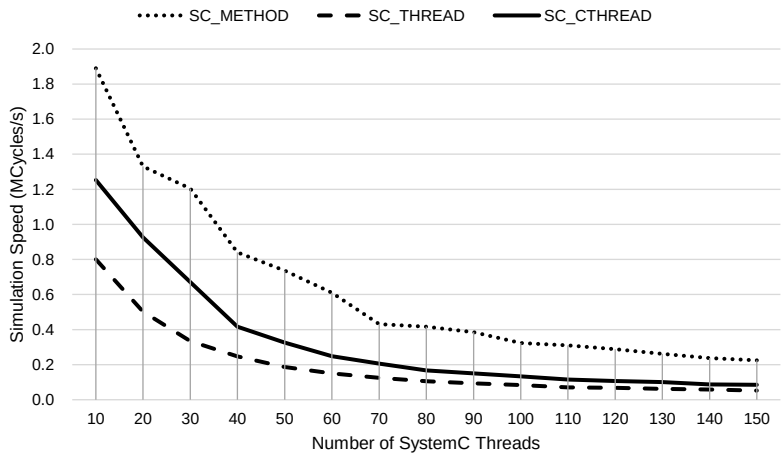


Figure 5.4: Simulation Performance of OSCI SystemC Simulator

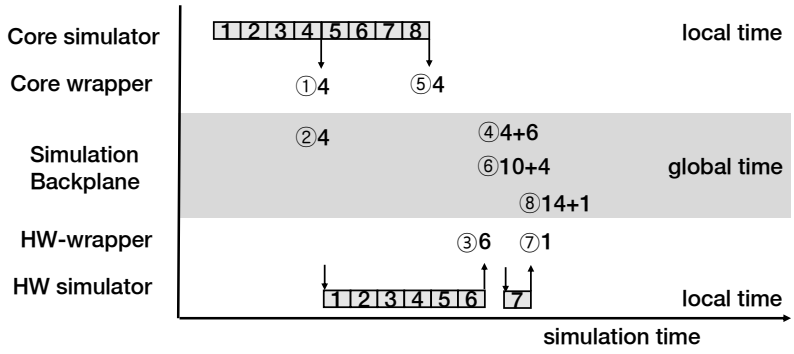


Figure 5.5: An Execution Scenario Based on Trace-Driven Co-Simulation Technique

5.4 Reducing the Cost of Interprocess Communications

5.4.1 Trace-driven Co-simulation

The many-core simulator assumed in this thesis basically follows the ISS-SystemC co-simulation framework. Each ISS is responsible for core simulation, while the simulation backplane simulates components other than cores, and arranges all events occurring in the system in chronological order. All ISSs and simulation backplanes are created and run in different processes. Since each OS process is prohibited from accessing the address space of another process, interprocess communication APIs are used for communication between processes.

The most important communication between processes is time synchronization. Time synchronization guarantees events occurring in each process to be simulated in the chronological order. If time synchronization is not properly performed, a causal error occurs when a past event occurs later than the current event, and the accuracy of the simulation is compromised.

We used trace-driven co-simulation which was proposed in [64][65] to reduce the number of interprocess communications. The beauty of this technique is that a core simulator can proceed without worrying about the global clock managed by the simulation backplane.

Basically, a core simulator performs IPC only when there is an access to the shared resources managed by the simulation backplane. In the case of the target architecture that we are assuming, L1 data cache access and L1 instruction cache miss generate events which access to the shared resources. Blocking IPC is used when the core simulator must wait for a response from the simulation backplane. Otherwise nonblocking IPC is used. For example, memory read requests use blocking IPC, while memory write requests use nonblocking IPC. Allowing nonblocking IPC, core simulators can minimize wasting time being blocked. Thus, the effect of parallel simulation is maximized.

Once IPC occurs, the core simulator sends the difference between the local clock of most recent message and current local clock. The simulation backplane updates the simulated time of the core wrapper using this delta time, and obtains the accurate timing of a event. Processing from the earliest event, the simulator ensures the strictly-conservative causality. On the other hand, in a situation where the shared resource is not accessed, the core simulator continuously simulates, advancing the local clock.

We presented the procedure in Figure 5.5. Processors, or their core simulators, run until they encounter shared resource accesses as describes in ①. The core sends the request, piggybacking the time difference of the current local time and the latest synchronization time, which is 4. The core wrapper receives and updates the current local time of the core simulator (②). The simulation backplane receives a request for shared resource from all cores and finds out that it is the earliest request. So the re-

quest is transferred and serviced to the target HW simulator and the local time of HW simulator is also updated (③) after the service. Meanwhile, if the request from the core simulator is nonblocking (ex. write operation), the core simulator advances its local time without waiting for the response (④).

The simplified code for the periodic main process of the core wrapper module is given in the following:

```
1  void QEMUCore::periodic_process()
2  {
3      while (true){
4          if (qemu->get_status() == RUNNING){
5              qemu->recv_packet(&packet);
6              wait(clock_cycle_time * packet.cycle);
7              handle_packet(&packet);
8          }
9          else if (qemu->get_status() == TERMINATED){
10             break;
11         }
12     }
13   ...
14 }
```

In line 5, the core wrapper does a blocking read from the core simulator, waiting a next message. In line 6, once a message is transferred to the wrapper, the core wrapper advances its local simulated time by the time difference piggybacked in the message. In line 7, the message is finally handled by the core wrapper.

5.4.2 Better Interprocess Communication

Inter-process communication (IPC) is a collection of programming interfaces that allow a programmer to communicate among different processes that can run concurrently in an operating system. UNIX provides several mechanisms for interprocess communications, each with their own benefits and tradeoffs. The representative IPCs are given as follows:

- Pipe: unidirectional data channel. using pipe()
- Socketpair: unnamed pair of connected sockets. using socketpair()

- Shared memory: shared memory segment. using shmget(), shmat()
- Tcp: communication based on the Internet protocol. using socket(), bind(), listen(), accept().

Most IPCs including pipe, socketpair, and tcp support blocking read and write, while shared memory segment does not. To use the shared memory segment for the packet transmission between ISS and simulation backplane, we implemented a ring buffer on the shared memory segment to accommodate multiple packets.

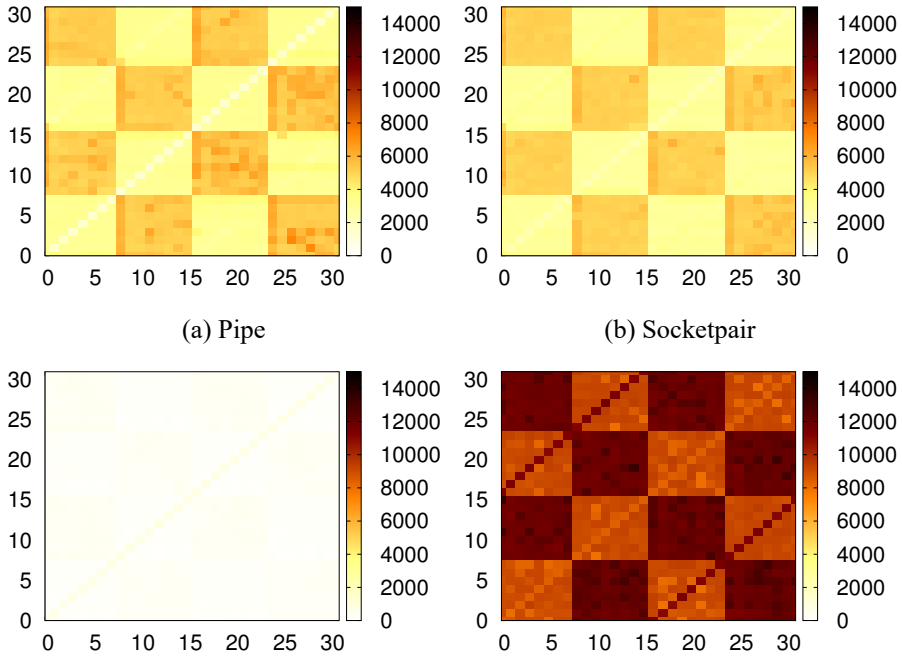
Inspired of [66], we evaluated and compared the performance of different IPC mechanisms. Two processes are generated by using fork() function, and then they are reallocated on various host cores using the following set_affinity() function. We measured the average packet latency of given IPCs between all senders and receivers.

```
1 void set_affinity(int cpuid) {
2     cpu_set_t set;
3     CPU_ZERO(&set);
4     CPU_SET(cpuid, &set);
5     printf("sched_affinity %d\n", cpuid);
6     if (sched_setaffinity(getpid(), sizeof(set), &set) == -1)
7         perror("sched_affinity");
8 }
```

In Figure 5.6, the experiment results confirm that IPC using the shared memory segment outperforms other approaches, showing only 300 ns latency. Therefore, we deployed the shared memory segment to our simulators as a sole IPC mechanism.

5.4.3 Virtually embedding modules to core simulator

Meanwhile, the underlying many-core architecture is not friendly to our ISS-SystemC and trace-driven co-simulation framework. This is because the L1 data cache, one of the most frequently accessed memory, is located on the SystemC backplane to be shared among the cores in the tile. Therefore, each core simulator is forced to do the IPC communication with the SystemC backplane every 0-2 simulated cycle, which is for memory read/write instructions. Eventually, the core simulator should spend most of their execution time waiting for the simulation backplane.



(a) Pipe

(b) Socketpair

(c) Shared Memory Segment

(d) TCP

IPC	pipe	socketpair	shm segment	tcp
Average	4322	4123	298	10575
Minimum	1444	2226	157	7912
Maximum	7277	6279	949	13449

Figure 5.6: Interprocess Communication Latency (ns) Measurement

However, certain memory area is exclusively used for each core. Using this fact, L1 cache can be virtually segmented into three caches: a private cache for the first core, a private cache for the second core, and a shared cache for both cores. Then, each private cache is embedded into the core simulator's process. If a memory access to a non-sharable memory space occurs, the core simulator performs the cache simulation for the address to rule the cache hit or miss. If a cache hit occurs, the core simulator can continue. Otherwise, the core simulator should resort to generate a message to the simulation backplane to access the shared cache. In this way, it is possible to dramatically reduce the number of IPC communications, while increasing the utilization of

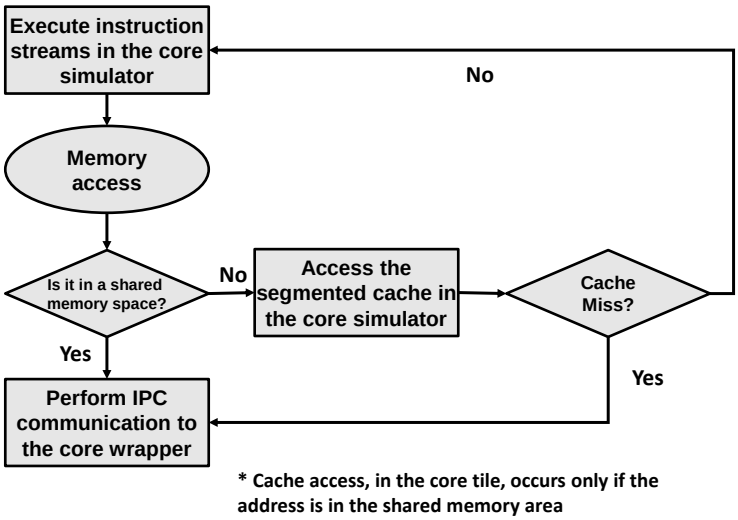


Figure 5.7: The Flowchart For a Core Simulator with the Virtually Segmented L1 Cache

core simulators. The flowchart is presented in Figure 5.7. Experiments show that this trick speeds up the simulation 16 - 64 times in terms of MIPS.

The term “virtually” is used because the actual target architecture has a single unified L1 data cache for each tile, and the unified cache is arbitrarily partitioned in order to improve the performance of the simulator. The cache statistics may subtly vary because of the different organizations of the cache structure.

5.5 Reducing SystemC Scheduling Overhead

5.5.1 Event-based Slave Module Activation

One of the most critical factors in the simulation speed in the SystemC scheduler is the number of activated modules. If the number of SystemC modules that have to be carried out per each simulated cycle increases, the performance of the sequential SystemC scheduler has been reduced linearly.

Suppose that slave module means a module that does not create a transaction by itself. Then, a core module is not a slave module, while other modules, such as tile, NoC, bus, and device memory are slave modules. Typically, once a master module

generates transactions and transmits them through a socket, slave modules receive and serve the transactions.

On the other hand, a slave module usually has a clock-sensitive process. For example, suppose that a cache miss occurs in a cache tile module. Then the cache tile module creates a memory fetch request to handle the miss event. In a similar way, the cache tile module generates a memory read request to handle a cache flush or write-back event. Those memory requests are managed by a clock-sensitive periodic task of the cache tile module. There are two ways to implement these clock sensitive functions as the SystemC process.

The first method is to declare the periodic process in the SC_METHOD and to make it sensitive to the positive edges (positive edges) in the clock. In this case, the periodic function will wake up every cycle to determine whether to work. This is the most intuitive approach, but at the same time, the periodic process must be woken up every single clock. Even though there is nothing to do, the periodic process wakes up and sleeps immediately.

The second method is to declare the periodic process in the SC_CTHREAD, and then make it wait and sleep for the appropriate event. In this case, the function is woken up at the first clock positive edge, and then the process becomes inactive regardless of the clock event until the event is notified. Meanwhile, the slave module is accessed only with callback functions and the corresponding payload processing. Once the workload for the periodic task becomes available due to a cache miss or flush, the event is finally notified to wake up the periodic process.

The simulation results are identical in both ways. However, the second method significantly can reduce the number of processes that are activated every cycle, which means that more important process (in the critical path) can be scheduled earlier. As a result, the second approach was to ensure that 4-11 times performance improvement when compared to the first approach.

5.5.2 SystemC Scheduler Parallelization

SystemC that is a collection of C++ classes and macros provides the simulation environment for transaction-level modeling [67]. SystemC supports various APIs to simulate hardware components that are running concurrently in reality. The behavior of each hardware component is modeled by a SystemC process, SystemC method or thread, which is executed by the SystemC scheduler. Care should be taken to preserve the chronological order of events to avoid any causality error during simulation.

A naive and simple scheme is to use a system-wise global event queue and servicing the events sequentially, which is taken in the baseline SystemC simulation kernel provided by the Open SystemC Initiative (OSCI) [63]. As result, the sequential OSCI reference simulator becomes the bottleneck of simulation for complex systems, and extensive researches have been conducted to parallelize SystemC simulation, mostly applying fully-synchronous approach[68]–[72]. Recently, a few literatures are published to introduce parallel distributed event scheduling approaches to parallelize SystemC simulation [73]–[75]. However, to the best of our knowledge, no previous work supports the approximately-timed coding style of OSCI Transaction Level Modeling standard 2.0 (TLM-2.0) model, which is introduced to increase model interoperability [3].

In this thesis, we propose a novel parallel distributed scheduling technique which can support the approximately-timed coding style of TLM-2.0 model. With a given mapping of modules to simulation host cores, the proposed technique performs time synchronization locally at each module without global time synchronization. To increase the degree of parallelism of simulation, the lookahead time by which a module can advance its local clock without time synchronization is computed at compile-time.

However, when a SystemC parallelization technique was applied to our many-core simulator, we found that there was little improvement in performance because of a lack of concurrent processes and excessive lock/cache overheads. Further details are discussed in the following chapter.

5.6 Evaluation

In this section, the proposed many-core simulation backplane is evaluated.

5.6.1 Scalability Test

For the scalability test, the performance of the simulation is measured by changing the number of host cores used for simulation. Two synthetic benchmarks, computation-intensive *synth1* and memory-intensive *synth2* are designed to activate all target cores of the target architecture. The experimental results are given in Table 5.1.

As indicated in Table 5.1, the significant performance improvement is observed as the number of host cores increases. In addition, when increasing the number of cores from 1 to 4, super-linear speed-up is observed. It is due to the cost of excessive context switching and interprocess communications inside a single host core.

5.6.2 Simulation Performance

Six realistic benchmarks were used to evaluate the performance of the proposed simulator. *Volume Rendering* is aa in-house benchmark to display a 2D projection of a 3D discretely sampled data set. *Volume Rendering* benchmark is parallelized for each pixel. *Prime* benchmark is another in-house benchmark to collect a set of prime numbers in a specific range. On the other hand, other four benchmarks are obtained from representative parallel benchmark suites: *FFT* and *LU* are ported from Splash2 suite [76], while *Blackscholes* and *Swaption* are ported from Parsec benchmark suite [77].

In the experiment, it is observed that the simulator performs at more than 500 KCycles/s with most realistic benchmarks.

Table 5.1: Full-System Simulation Scalability Test(MIPS)

Number of host cores	Computation-intensive Synthetic	Memory-intensive Synthetic
1	0.38	0.05
4	3.16	1.23
16	16.6	4.12
32 (hyper-threading)	26.8	5

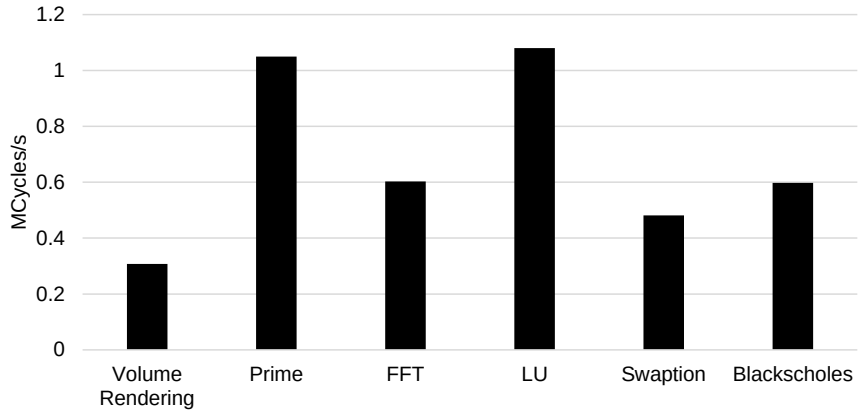


Figure 5.8: Simulation Performance (MCycles/s)

5.6.3 Simulation Accuracy

The proposed simulation backplane is a cycle-accurate based on conservative synchronization. Thus, since there is neither approximation nor causality error introduced by the system model, the estimation of performance figures is completely error free if events generated from component simulators have exact timing. Thus, we believe that our simulation backplane can be a good reference simulator to be compared with other performance-first techniques at the cost of accuracy.

Except core simulation, cache and NoC simulations mainly influence the precision of the simulation. First, we observed that the NoC model included in the proposed simulator presents identical correct statistics generated from Noxim simulator. This is because our NoC simulator was written on the basis of Noxim code. Second, we provide a set of cache traces to our cache simulator and validated state-of-the-arts cache simulator Dinero IV[78], and observed that they generate identical cache statistics. Since our target architecture does not support hardware cache coherency, we judged that this degree of verification is reliable enough.

Chapter 6

Simulation Backplane Parallelization

6.1 Background: OSCI SystemC Scheduler

Figure 6.1 shows the structure of the OSCI SystemC scheduler. Processes with the same timestamp are evaluated based on the cooperative multitasking mechanism where a process continues to execute until it voluntarily gives up the control once started. If there are no more processes to execute, channels are updated based on the results of the evaluations and the system clock is incremented by a delta cycle that is used to order simultaneous events by data dependencies. Note that the time notion of SystemC is defined as a tuple (wall clock time, delta cycle). After incrementing a delta cycle, processes which become runnable by channel updates are evaluated. This iterative process ends when there are no more runnable processes after channel updates. Then the delta cycle is initialized to 0, and the wall clock time advances to the earliest timestamp of the existing processes in the sorted queue. The simulation continues until there remain no more processes in the system.

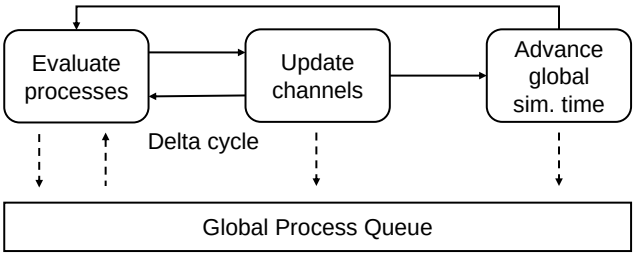


Figure 6.1: The Structure of the Scheduler of OSCI Implementation

6.2 Related Work: SystemC Parallelization Techniques

Extensive researches have been performed to parallelize SystemC simulation for boosting up the simulation speed. They are briefly reviewed and compared with the proposed approach in this section.

6.2.1 Fully-synchronous Approach

An intuitive method is to execute processes at the same delta cycle in parallel without any risk of causality error. Techniques in [68]–[72] belong to this category, named fully-synchronous approach. In this category, paralleling the SystemC simulation using GPGPUs has been studied in [79], [80]. Note that the synchronization point of this approach is every delta cycle. They are effective only when there exist many runnable processes at each delta cycle. If the timestamps of runnable processes are different from each other in the worst case, this approach degenerates to sequential simulation.

6.2.2 Parallel Distributed Event Scheduling (PDES) Approach

To reduce the excessive synchronization overhead of the fully-synchronous approach, several techniques have been proposed to adopt the parallel distributed event scheduling (PDES) approach where time synchronization is performed locally at each module without global time synchronization. The proposed technique belongs to this category.

The authors of [73], [74] suggested a new programming model, named TLM-DT (Distributed Timed), in which each process manages a local time and sends it as the

extra argument of the transport interface methods. TLM-DT requires us to explicitly specify time management in the simulation model, which is assumed to done by the SystemC scheduler in TLM-2.0 standard.

Recently, a time-decoupled parallel SystemC simulation technique has been proposed by adopting the notion of lookahead [75]. Each kernel thread owns its local time and may advance to a certain time limit without synchronization. The time limit is determined by the minimum of the local times of the other threads plus the lookahead. After a thread reaches this limit, it computes a new limit. Thus, the lookahead is a key parameter that affects the number of synchronization points in this approach. The lookahead value is given a priori and applied globally to all modules, they assume. To avoid causality errors, any notifications that have a shorter notification time than the lookahead are strictly forbidden. [75] supports the TLM-LT coding style only, if synchronization is explicitly specified by sc_event. On the other hand, the proposed technique computes the lookahead time between each pair of modules individually. In addition, the proposed technique supports both TLM-LT and TLM-AT coding styles and assumes implicit synchronization using payload event queues, as recommended by the TLM-2.0 standard.

6.2.3 Out-of-order Execution with Dependency Analysis

The PDES approach still needs to process the events in the chronological order in each core. If two events A and B have no causality relation, however, they can be processed in any order or in parallel. The authors of [81] proposed a technique to perform compile-time dependency analysis of the threads to enable out-of-order execution of events. Running data-independent threads out-of-order and in parallel maximizes the benefit of parallel simulation. The static analysis is applied to the segment graph, which is generated from the simulation source code at compile time. They extended the approach to achieve better simulation performance by avoiding false conflict with prediction [82]. While this approach parallelizes the SystemC model at the event level, the proposed approach and the PDES approach assume that mapping and partition-

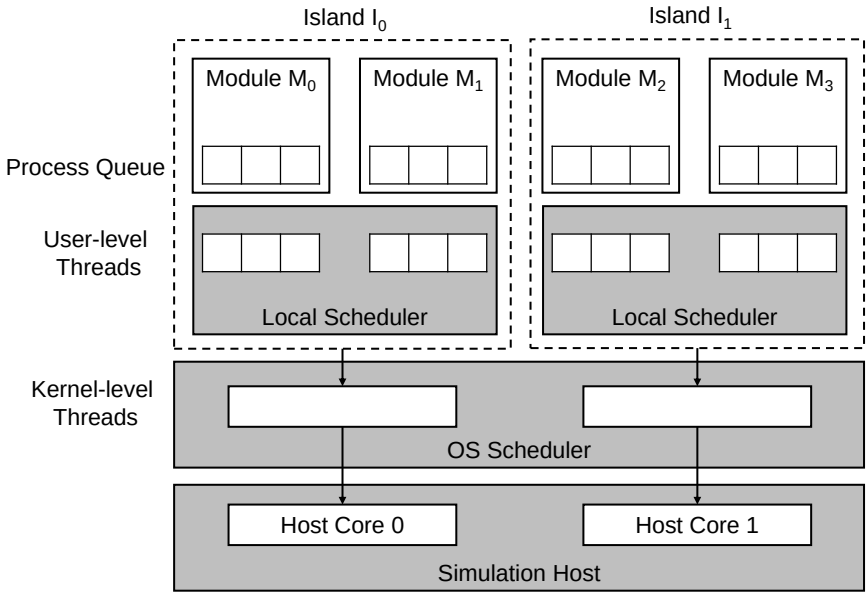


Figure 6.2: The Thread Organization of the Proposed Scheduler

ing of modules to cores are decided manually. The reported speed-up by automatic parallelization is somewhat limited despite sophisticated compile time analysis.

6.2.4 Dynamic Offloading Approach

An approach proposed in [83] creates a thread dynamically that performs a specified computation defined in a new non-standard primitive *sc_during*. Semantically, the dynamically created thread can be executed in parallel with the rest of the simulation. Dynamic offloading may degrade the simulation performance if the offloaded computation is not long enough to compensate the run-time overhead.

6.3 Proposed Technique

The proposed technique uses two-level multi-threading to parallelize the SystemC model, as shown in Figure 6.2. Each kernel thread is allocated to a host core, and executed in parallel. Suppose that the system, $\mathbb{M}$, is composed of a set of m modules: $\{M_0, M_1, ..., M_{m-1}\} = \mathbb{M}$. A process in a module is implemented as a user-level thread like OSCI implementation. Each module is assigned to one kernel thread stati-

cally by the simulation model designer who is assumed to have deep knowledge of the communication patterns between modules. Re-assigning a module to another kernel at runtime is not an impossible task in the proposed technique, which is left as a future work.

A set of modules assigned to the same kernel thread is called an island. There are n islands $\{I_0, I_1, ..., I_{n-1}\} = \mathbb{I}$, where n is the number of host cores in the simulation host. Each module is mapped to one island by $mapping : \mathbb{M} \mapsto \mathbb{I}$. Each island owns a local scheduler, which performs cooperative scheduling of processes assigned to the island. Figure 6.3 displays the structure of the local scheduler, which is seemingly similar to the OSCI implementation of Figure 6.1. Processes in the island are executed in a fully-synchronous way.

While OSCI implementation manages a global clock, the proposed techniques manage the local clocks of modules in a distributed way. Module M_k manages its local time lt_k and the earliest timestamp of processes nt_k. When a module requests the current simulated time (by *sc_time_stamp()* API), the local time of the module is returned. A list of local times and earliest event times of modules is stored in the shared memory space, which can be accessed by all kernel threads without explicit message passing. To avoid the false sharing problem [84], the time information has to be carefully aligned. Note that when the earliest process of module M_k begins to execute, local time lt_k is advanced to nt_k, and a new nt_k is determined. nt_k is updated each time the module's process queue is updated and the process schedules the earliest process, while lt_k is updated every time the process schedules a process.

6.3.1 Basic Synchronization

As shown in Figure 6.3, the local scheduler maintains the process queue of modules in the island. The basic synchronization scheme is to guarantee that processes in the same island are scheduled in the time order and to avoid any causality error. The causality error in module M_k occurs if the next earliest-event timestamp nt_k is earlier than the current local time lt_k ($nt_k > lt_k$).

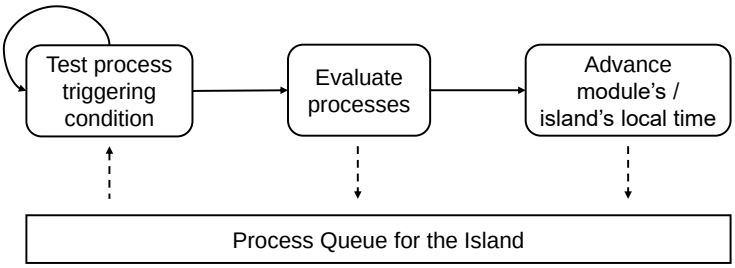


Figure 6.3: The Structure of the Local Scheduler of the Proposed Approach

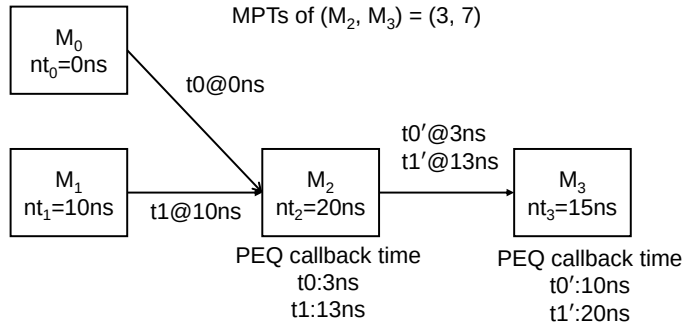
Remind that in TLM-2.0 communication between modules takes place via function calls. When a initiator module calls a nonblocking transport function, it lets the target module register the payload of the transaction with the processing time to the payload event queue. The scheduling of a process must be delayed until it is guaranteed that no earlier transaction will be requested from a remote island. Hence, a process should wait until it becomes the earliest of the whole system, preventing parallel "out of order" execution. To execute the earliest process of M_k in an island safely, the following process triggering condition should be satisfied: $nt_k \leq nt_i$ for $\forall M_i, mapping(M_i) \neq mapping(M_k)$.

6.3.2 Relaxed Synchronization

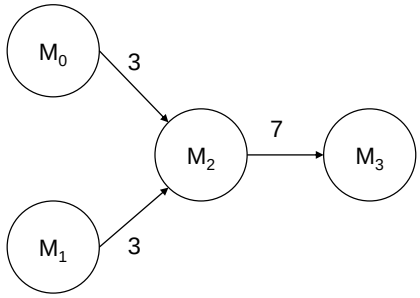
In the basic synchronization scheme, the earliest process of an island should wait until it becomes the earliest over the system, which severely restricts parallel simulation. To relax this tight synchronization condition, the following two schemes are devised in the proposed technique.

First, for *initiator modules* that have only initiator sockets, testing the process triggering condition can be exempted if the module does not require the target module to send any response. For example, If the initiator is about to schedule a write transaction that does not need to wait for a result, it is possible to schedule it without passing the process triggering condition. To implement this scheme, the non-blocking transport functions are redefined to notify the local scheduler of such a case.

Second, we compute the lookahead time that a module can advance its local clock



(a) A Simulation Model Where "PEQ" Stands For Payload Event Queue



(b) Latency Graph Generated from the Model

i \ j	0	1	2	3
0	0		3	10
1		0	3	10
2			0	7
3				0

(c) Lookahead Table Obtained from the Latency Graph

Figure 6.4: Lookahead Computation

without synchronization safely. Unlike the lookahead time that is defined for each kernel thread in [75], our lookahead time is defined between two inter-dependent modules.

Figure 6.4 illustrates how lookahead computation is performed and used. Suppose that the earliest process times of module M_0, M_1, M_2, and M_3 are 0, 10, 20, and 15 ns in the simulation model of Figure 6.4 (a). The basic synchronization scheme cannot execute the process of M_3 until the earliest process times of other modules are equal to or bigger than 15 ns.

Suppose that the minimum payload processing times of M_2 and M_3 are 3 and 7 ns, respectively. The payload processing time means the time required for the module to process the payload from other modules. If module M_0 sends transaction t_0 to module M_2 at 0 ns, it will process t_0 at 0+3 ns and send transaction t_0' to module M_3 at 0+3 ns. Then the transaction will be processed at 3+7 ns at module M_3. Since the earliest

process time of module M_3 is 15 ns, the module M_3 should wait until it receives a transaction from M_0 (through M_2). On the other hand, transaction t_1 from M_1 will affect module M_3 at 20 ns at the earliest. Thus, the module M_3 can execute the process of 15 ns not waiting for module M_1. The minimum latency of an effect chain from module M_i to module M_k is defined as lookahead $lookahead_{i,k}$.

Once we obtain the lookahead between modules, the process triggering condition can be relaxed to the following:

The earliest process of M_k can be scheduled if $nt_k \leq nt_i + lookahead_{i,k}$ for $\forall M_i, mapping(M_i) \neq mapping(M_k)$

For lookahead computation, the minimum payload processing time (MPT) of each module is required. To this end, it is recommended to annotate each module with the following preprocessor directive that does not harm the portability of the SystemC model: *#pragma tlm2.0 min_processing_time(sc_time)*. Since a module usually has an internal latency required to process the payload from other modules, it can be done easily by the user. Based on the topology of the system and the MPT information, the scheduler builds a latency graph as shown in Figure 6.4(b). The weight of an edge (M_i, M_j) is determined by the MPT of the destination module M_j. Then the lookahead table element between two modules is obtained by any algorithm for finding the shortest paths between nodes in a graph such as the Floyd–Warshall algorithm [85].

6.3.3 Modeling Restrictions

Since the proposed simulation technique is devised to support the coding styles that the TLM-2.0 reference manual recommends, the current implementation has the following restrictions on model definition.

- Using an event for communication between modules in the different islands is not allowed. Internal events that are declared and used inside one island, however, are allowed.

- Time-sensitive action may not be defined in the callback functions of a non-blocking transport function but should be handled by registering to the payload

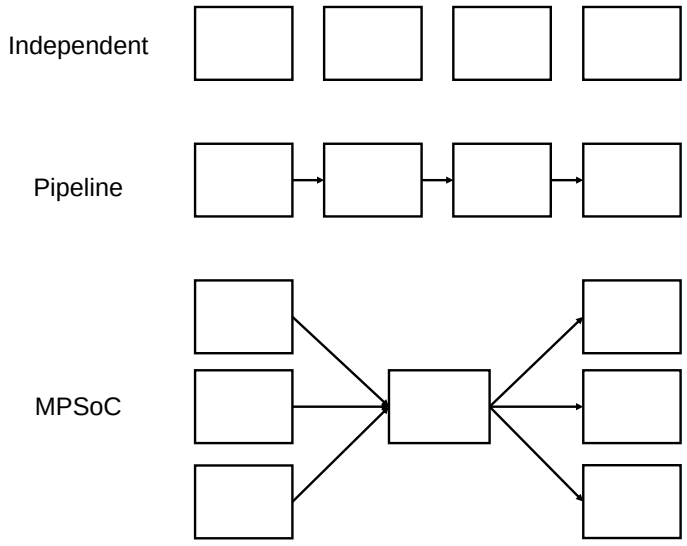


Figure 6.5: Benchmark Topologies

event queue. Note that the callback function in a target module is executed in the context of the initiator module.

6.4 Experimental Results

To validate the effectiveness of the proposed parallelization and relaxed synchronization technique, we compare the following four scheduling techniques: 1) OSCI implementation, 2) a fully-synchronous simulator that synchronizes the kernel threads at every time advancement 3) the proposed parallel scheduler without relaxed synchronization, and 4) the proposed scheduler with relaxed synchronization. The simulation host machine is an Ubuntu Linux 14.04 64-bit system with two Xeon E5-2640V2 processors clocked at 2.00GHz, allowing concurrent scheduling of 32 threads with hyper-threading.

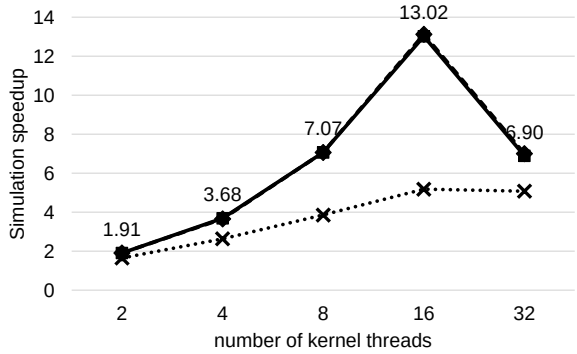
Experiments are conducted with a set of benchmarks with three different topologies as illustrated in Figure 6.5: *Independent*, *Pipeline*, and *MPSoC*. *Independent* benchmark contains 100 independent modules that do not communicate with each other. It is evident that the benchmark will get the most benefit from parallel simulation and

give the upper bound of the speed-up achievable from any parallelization technique. In *Pipeline* benchmark, the initiator module in the first stage generates transactions every 10 ns. The transaction is transferred to the next module and processed. This procedure is repeated until all transactions are delivered to the end of pipeline. The depth of pipeline of the benchmark is 100, and the MPT of each module is identically set to 10 ns. The *MPSoC* benchmark describes a multicore system that consists of multiple master and slave modules with a central interconnection module. Each master (processor core) generates a sequence of memory read/write requests to the interconnection module. Then the interconnection module routes the transaction to a slave module (memory) based on the address of the transaction. The slave module serves the transaction, and returns the transaction to the initiator of the transaction through the interconnection. Since TLM-2.0 standard is designed for transaction-level memory-mapped bus modeling, this benchmark is more practical than the other two. The numbers of cores/slaves are 10. Initiator modules generate the transaction every 100 ns, and the MPT of the interconnection module is set to 100-200 ns, based on the transaction route. The MPT of each slave module is 200 ns. For all benchmarks, the wall-clock execution time of a single execution of each process is randomly chosen from 1 us to 5 ms.

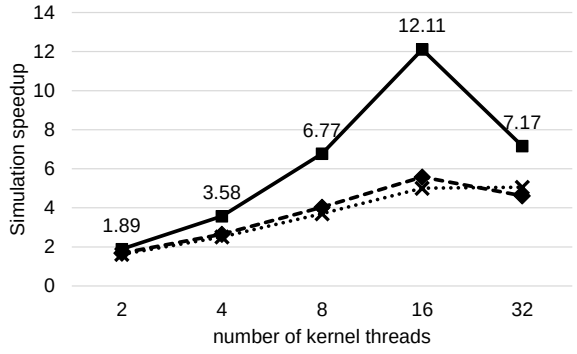
6.4.1 Performance

We first measure the simulation time varying the number of simulation hosts which is equal to the number of kernel threads. Figure 6.6 shows the speed-up of three other techniques compared with the OSCI implementation. The results are obtained by taking the average of 30 instances for each type of benchmark.

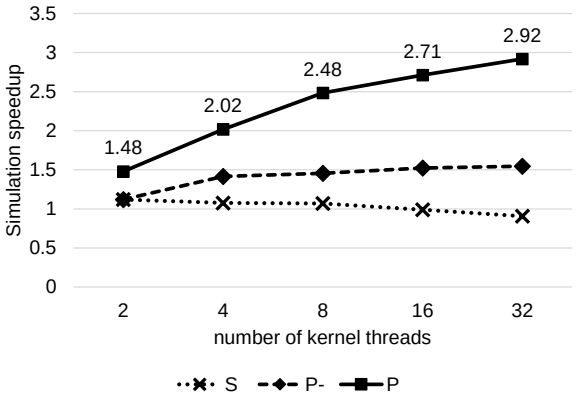
For *Independent* benchmark, the proposed simulation technique shows almost linear speed-up with the number of host cores, as expected. However, the fully synchronous scheduler pays the huge overhead of time synchronization at every clock advancement to result in much smaller slope of speed-up. The *Pipeline* benchmark shows the benefit of relaxed synchronization clearly. Since all modules are dependent in this benchmark, the local scheduler of each island should examine the local



(a) *Independent*



(b) *Pipeline*



(c) *MPSoC*

S : a fully synchronous scheduling technique
P- : the proposed parallel scheduling technique without relaxed synchronization
P : the proposed parallel scheduling technique with relaxed synchronization

Figure 6.6: Speed-Up of Various Approaches Compared to the OSCI Implementation

clocks of all the other islands, paying the similar synchronization cost as the fully synchronous scheduler. Thus, the performance gain from the proposed scheduler without relaxed synchronization, which is 5.58 (16 kernel threads), is similar to that from the fully synchronous scheduler. On the other hand, the relaxed synchronization technique successfully could accelerate parallel simulation to achieve 12 times speed-up.

The proposed technique with related synchronization achieves 2.7 times speed-up with 16 host cores for the *MPSoC* benchmark. In this benchmark, modules are dependent and the dependency path is so short (2 hops at most) that the gain from the lookahead computation is not significant. In other words, the potential parallelism of the benchmark is not large compared with the other types of benchmarks. Notwithstanding, the proposed technique with relaxed synchronization shows much better speed-up and scalability than the fully synchronous scheduler and the parallel scheduler without relaxed synchronization. Since there is little performance gain from 8 cores to 16 cores, it is better to use up to 8 cores for the *MPSoC*-type benchmark. Note that the fully synchronous scheduling technique results in negative speed-up for this benchmark since there is little work to parallelize while paying the same synchronization overhead as the other benchmark case.

In all cases, the performance advantage of the proposed technique with relaxed synchronization becomes evident as the number of kernel threads increases.

6.4.2 Accuracy

To verify the accuracy of the parallel simulation technique, the following three checklists are used:

- The final simulated time should be identical with the OSCI implementation.
- The numbers and timestamps of transactions generated from the proposed implementation should be identical to those from the OSCI implementation.
- No causality error is observed.

Since the proposed scheduler passes all checklists, the accuracy of the proposed simulator is empirically confirmed.

6.5 Discussion and Limitation

In this chapter, we present a novel parallel distributed scheduling technique which supports the approximately-timed coding style of TLM-2.0 model, which has not been addressed in the previous work to the best of our knowledge. With a given mapping of modules to simulation host cores, the proposed technique performs time synchronization locally at each module without global time synchronization. To increase the degree of parallelism of simulation, the relaxed synchronization technique is devised. By computing the lookahead time between each pair of modules we may advance the local clock of a module without time synchronization. Experimental results confirm the benefit of the proposed parallelization and relaxed synchronization technique, compared with the fully synchronous and the technique without relaxed synchronization. In this work, we assume that the mapping of modules is given a priori. Since the load balancing between host cores is an important factor that affects the simulation performance, dynamic load balancing will be a future research topic.

However, when the proposed techniques were applied to our many-core simulator, we found that there was little improvement in performance. There are several reasons:

- **Lack of concurrent processes at a single cycle:** The proposed TLM-level many-core simulator does not have enough number of concurrent processes at a single cycle. Note that parallelization is generally more effective as the number of processes at the same cycle is large enough.

- **Existence of the interconnection module:** The proposed TLM-level many-core simulator has a shallow structure with the interconnect module, which forces centralized synchronizations. This is why the benchmark *MPSoC* does not have a lot of performance benefits.

- **Short computation time of each process:** If executing a process takes a

very short period of time, parallelization gain would be smaller than lock and cache/memory overheads.

- **Lock acquisition overhead:** In the approximately timed model we assume, the communication between modules is achieved by registering a process to the connected module. If such modules are allocated to different hardware threads, push and pulling a process to the queue require the current thread to acquire the lock. To be more specific, each module is required to acquire the lock to access the payload event queue of the other module. In addition, if the module is connected to other module in another thread, the module itself should acquire the lock to pull and schedule the earliest-timed process. The locking overhead becomes more significant as a process takes a very short period of time.

- **Memory/cache access overhead:** Testing triggering conditions incurs excessive memory accesses, producing a sequence of read requests to check earliest time-stamps of all related modules. Once a module updates its own earliest time-stamp, cache copies of the same memory location in other host cores must be invalidated, and massive memory accesses are incurred.

Chapter 7

Conclusion

In this dissertation, we have extensively reviewed existing research associated with many-core simulation techniques. We establish a fast and accurate simulation framework for many-core NoC architecture.

The dissertation focuses on three main components: core simulation, NoC simulation, and the parallel simulation backplane. For core simulation, analytical (interval) + sampled hybrid simulation model was proposed, and validated with a cycle-accurate simulator. The proposed core simulator shows 11.36 to 44.31 MIPS performance, while simulation error remains less than 8 percent. For NoC simulation, we propose detailed flit-level NoC simulation to model buffer- and link-congestions. Applying software optimizations such as skipping idle modules and using packet/flit pool, the NoC simulation performance is at least two times faster than other state-of-the-arts. The proposed parallel simulation backplane supports conservative synchronization, which is free of causality errors. We presented several techniques to reduce interprocess communication overheads and scheduling overheads. We proposed parallel SystemC scheduler, but we observed that parallel SystemC scheduler bring little improvement in performance, and analyzed why.

The current simulation speed is up to 32 MIPS if the workload is well distributed

to utilize all cores, and above 500KCycles/s for most realistic applications.

As future works, we want to try various simulation techniques for NoC. The higher the number of cores, the more likely the NoC will become a bottleneck. The candidates are packet-level simulation, analytical model, or FPGA-based Parallelization. The accuracy validation of the proposed simulator was very limited, so it is required to improve and validate the accuracy of simulation by using development boards including NoC. Although the simulation target architecture does not support hardware-level cache coherency, it would be beneficial for generality of the simulator to study optimization technique to provide efficient cache coherence layer. However, we believe that SystemC Parallelization would not be promising for TLM-level modeling

Bibliography

[1] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith, "The future of simulation: A field of dreams," *Computer*, vol. 39, no. 11, pp. 22–29, 2006.

[2] F. Bellard, "QEMU , a Fast and Portable Dynamic Translator," *Proceedings of the USENIX Annual Technical Conference.*, pp. 41–46, 2005.

[3] O. S. Initiative, "Osci tlm-2.0 language reference manual," *Version JA32, http://www. systemc. org*, 2009.

[4] N. Binkert *et al.*, "The gem5 Simulator," *Computer Architecture News*, vol. 39, no. 2, p. 1, 2011.

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The m5 simulator: Modeling networked systems," *IEEE Micro*, vol. 26, no. 4, pp. 52–60, 2006.

[6] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 92–99, 2005.

[7] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, Institute of Electrical and Electronics Engineers (IEEE), 2009.

[8] A. Patel, F. Afram, S. Chen, and K. Ghose, “MARSS: a full system simulator for multicore x86 CPUs,” in *Proceedings of the 48th Design Automation Conference*, ACM, 2011, pp. 1050–1055.

[9] M. T. Yourst, “Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator,” in *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, IEEE, 2007, pp. 23–34.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, “Xen and the art of virtualization,” in *ACM SIGOPS Operating Systems Review*, ACM, vol. 37, 2003, pp. 164–177.

[11] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, “Dramsim: A memory system simulator,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100–107, 2005.

[12] T. E. Carlson, W. Heirman, and L. Eeckhout, “Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ACM, 2011, p. 52.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, “Pin: Building customized program analysis tools with dynamic instrumentation,” in *ACM Sigplan Notices*, ACM, vol. 40, 2005, pp. 190–200.

[14] T. S. Karkhanis and J. E. Smith, “A first-order superscalar processor model,” in *ACM SIGARCH Computer Architecture News*, IEEE Computer Society, vol. 32, 2004, p. 338.

[15] S. Eyerman, L. Eeckhout, and K. De Bosschere, “Efficient design space exploration of high performance embedded out-of-order processors,” in *Proceedings of the conference on Design, automation and test in Europe*, European Design and Automation Association, 2006, pp. 351–356.

[16] D. Genbrugge, S. Eyerman, and L. Eeckhout, “Interval simulation: Raising the level of abstraction in architectural simulation,” in *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, IEEE, 2010, pp. 1–12.

[17] D. Sanchez and C. Kozyrakis, “ZSim: fast and accurate microarchitectural simulation of thousand-core systems,” in *ACM SIGARCH Computer Architecture News*, ACM, vol. 41, 2013, pp. 475–486.

[18] J. Wang *et al.*, “Manifold: A parallel simulation framework for multicore systems,” in *IEEE International Symposium on Performance Analysis of Systems and Software*, IEEE, 2014, pp. 106–115.

[19] C. D. Kersey, A. Rodrigues, and S. Yalamanchili, “A universal parallel front-end for execution driven microarchitecture simulation,” in *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, ACM, 2012, pp. 25–32.

[20] G. H. Loh, S. Subramaniam, and Y. Xie, “Zesto: A cycle-level simulator for highly detailed microarchitecture exploration,” in *IEEE International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009*, IEEE, 2009, pp. 53–64.

[21] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.

[22] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas, “HORNET: A cycle-level multicore simulator,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 6, pp. 890–903, 2012.

[23] M. Lis, K. S. Shim, M. H. Cho, P. Ren, O. Khan, and S. Devadas, “Darsim: A parallel cycle-level noc simulator,” in *MoBS 2010-Sixth Annual Workshop on Modeling, Benchmarking and Simulation*, 2010.

[24] *OVP Homepage*. [Online]. Available: `http://www.ovpworld.org/technology{\_}ovpsim`.

[25] N Romdan, “ARM FastModels–Virtual Platforms for Embedded Software Development,” *Information Quarterly Magazine*, vol. 7, no. 4, pp. 33–36, 2008.

[26] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, “Efficiently exploring architectural design spaces via predictive modeling,” in *Proceedings*

of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 41, 2006, pp. 195–206.

[27] T. M. Conte, M. A. Hirsch, and K. N. Menezes, "Reducing state loss for effective trace sampling of superscalar processors," in *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers and Processors*, IEEE, 1996, pp. 468–477.

[28] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, IEEE, 2003, pp. 84–95.

[29] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5, pp. 45–57, 2002.

[30] S. Nussbaum and J. Smith, "Modeling superscalar processors via statistical simulation," *Proceedings 2001 International Conference on Parallel Architectures and Compilation Techniques*, pp. 15–24, 2001.

[31] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. De Bosschere, "Statistical simulation: Adding efficiency to the computer designer's toolbox," *Ieee Micro*, vol. 23, no. 5, pp. 26–38, 2003.

[32] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "Fpga-accelerated simulation technologies (fast): Fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture*, IEEE Computer Society, 2007, pp. 249–261.

[33] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research accelerator for multiple processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.

[34] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-based high-detail multicore simulation using time-division multiplexing," in *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture*, IEEE, 2011, pp. 406–417.

[35] S. Kraemer, L. Gao, J. Weinstock, R. Leupers, G. Ascheid, and H. Meyr, "HySim: a fast simulation framework for embedded software development," in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, IEEE, 2007, pp. 75–80.

[36] L. G. Murillo, J. Eusse, J. Jovic, S. Yakoushkin, R. Leupers, and G. Ascheid, "Synchronization for hybrid MPSoC full-system simulation," in *Proceedings of the 49th Design Automation Conference*, IEEE, 2012, pp. 121–126.

[37] R. Plyaskin and A. Herkersdorf, "Context-aware compiled simulation of out-of-order processor behavior based on atomic traces," in *VLSI and System-on-Chip, 2011 IEEE/IFIP 19th International Conference on*, IEEE, 2011, pp. 386–391.

[38] S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, and O. Bringmann, "Context-sensitive Timing Simulation of Binary Embedded Software," in *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2014, 14:1–14:10.

[39] S Stattelmann, S Ottlik, A Viehl, O Bringmann, and W Rosenstiel, "Combining instruction set simulation and WCET analysis for embedded software performance estimation," in *7th IEEE International Symposium on Industrial Embedded Systems)*, 2012, pp. 295–298.

[40] H. Cassé and P. Sainrat, "OTAWA, a framework for experimenting WCET computations," in *3rd European Congress on Embedded Real-Time Software*, 2006.

[41] C. Ferdinand and R. Heckmann, "ait: Worst-case execution time prediction by static program analysis," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, Springer, 2004, pp. 377–383.

[42] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout, "An evaluation of high-level mechanistic core models," *ACM Transactions on Architecture and Code Optimization*, vol. 11, no. 3, p. 28, 2014.

[43] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *IEEE International Symposium on Performance Analysis of Systems and Software*, IEEE, 2014, pp. 13–22.

[44] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop on Workload Characterization*, IEEE, 2001, pp. 3–14.

[45] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, "Virtual channels in networks on chip: Implementation and evaluation on hermes noc," in *Proceedings of the 18th annual symposium on Integrated circuits and system design*, ACM, 2005, pp. 178–183.

[46] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An Open, Extensible and Cycle-accurate Network on Chip Simulator," in *IEEE International Conference on Application-specific Systems, Architectures and Processors 2015*, 2015.

[47] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Cycle-accurate network on chip simulation with noxim," *ACM Transactions on Modeling and Computer Simulation*, vol. 27, no. 1, pp. 1–25, 2016.

[48] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software,*, IEEE, 2013, pp. 86–96.

[49] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: Simplifying event-driven programming of memory-constrained embedded systems," in *Proceedings of the 4th international conference on Embedded networked sensor systems*, Acm, 2006, pp. 29–42.

[50] N. Nikitin and J. Cortadella, "A performance analytical model for network-on-chip with constant service time routers," in *Proceedings of the 2009 International Conference on Computer-Aided Design*, ACM, 2009, pp. 571–578.

[51] U. Y. Ogras and R. Marculescu, "Analytical router modeling for networks-on-chip performance analysis," in *2007 Design, Automation & Test in Europe Conference & Exhibition*, IEEE, 2007, pp. 1–6.

[52] Z. Qian, D.-C. Juan, P. Bogdan, C.-Y. Tsui, D. Marculescu, and R. Marculescu, "Svr-noc: A performance analysis tool for network-on-chips using learning-

based support vector regression model," in *Proceedings of the Conference on Design, Automation and Test in Europe*, EDA Consortium, 2013, pp. 354–357.

[53] R. E. Bryant, "Simulation of packet communication architecture computer systems," 1977.

[54] K. M. Chandy and J. Misra, "Distributed simulation: A case study in design and verification of distributed programs," *IEEE Transactions on software engineering*, no. 5, pp. 440–452, 1979.

[55] B. D. Lubachevsky, "Efficient distributed event-driven simulations of multiple-loop networks," *Communications of the ACM*, vol. 32, no. 1, pp. 111–123, 1989.

[56] D. M. Nicol, C. C. Michael, and P. Inouye, "Efficient aggregation of multiple pls in distributed memory parallel simulations," in *Proceedings of the 21st conference on Winter simulation*, ACM, 1989, pp. 680–685.

[57] D. R. Jefferson, "Virtual time," *ACM Transactions on Programming Languages and Systems*, vol. 7, no. 3, pp. 404–425, 1985.

[58] B. Samadi, "Distributed simulation, algorithms and performance analysis," 1985.

[59] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Journal of Parallel and Distributed Computing*, vol. 18, no. 4, pp. 423–434, 1993.

[60] J. Chen, M. Annavaram, and M. Dubois, "SlackSim: A Platform for Parallel Simulations of CMPs on CMPs," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 2, p. 20, 2009.

[61] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal, "Graphite: A distributed parallel simulator for multicores," in *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, IEEE, 2010, pp. 1–12.

[62] R. M. Fujimoto, *Parallel and distributed simulation systems*. Wiley New York, 2000, vol. 300.

[63] Accellera Systems Initiative. (2014). Osci systemc 2.3.1, [Online]. Available: http://accellera.org/downloads/standards/systemc (visited on 09/14/2016).

[64] H. Kim, D. Yun, and S. Ha, "Scalable and Retargetable Simulation Techniques for Multiprocessor Systems," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2009, pp. 89–98.

[65] D. Yun, J. Kim, S. Kim, and S. Ha, "Simulation environment configuration for parallel simulation of multicore embedded systems," in *Proceedings of the 48th Design Automation Conference*, ACM, 2011, pp. 345–350.

[66] S. Smith and A. Madhavapeddy, "Draft: Have you checked your IPC performance lately?" *Anil.Recoil.Org*,

[67] *IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual*, 2006, pp. 0–423.

[68] B. Chopard, P. Combes, and J. Zory, "A conservative approach to SystemC parallelization," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 3994 LNCS, pp. 653–660, 2006.

[69] P. Combes, E. Caron, F. Desprez, B. Chopard, and J. Zory, "Relaxing synchronization in a parallel SystemC kernel," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, 2008, pp. 180–187.

[70] C Schumacher, R Leupers, D Petras, and A Hoffmann, "parSC: Synchronous parallel SystemC simulation on multi-core host architectures," in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2010, pp. 241–246.

[71] R. Dömer, W. Chen, and X. Han, "Parallel discrete event simulation of Transaction Level Models," in *Proceedings of the Asia and South Pacific Design Automation Conference*, 2012, pp. 227–231.

[72] M. K. Chung, J. K. Kim, and S. Ryu, “SimParallel: A high performance parallel SystemC simulator using hierarchical multi-threading,” in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2014, pp. 1472–1475.

[73] E. Viaud, F. Pecheux, and a. Greiner, “An Efficient TLM/T Modeling and Simulation Environment Based on Conservative Parallel Discrete Event Principles,” in *Proceedings of the Design Automation & Test in Europe Conference*, 2006, pp. 1–6.

[74] A. Mello, I. Maia, A. Greiner, F. Pecheux, I. M. aind A. Greiner, and F. Pecheux, “Parallel Simulation of SystemC TLM 2.0 Compliant MPSoC on SMP Workstations,” in *Proceedings of Design, Automation and Test in Europe*, 2010, pp. 606–609.

[75] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, “Time-Decoupled Parallel SystemC Simulation,” in *Proceedings of the Design, Automation & Test in Europe*, 2014.

[76] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The splash-2 programs: Characterization and methodological considerations,” in *ACM SIGARCH Computer Architecture News*, ACM, vol. 23, 1995, pp. 24–36.

[77] C. Bienia, S. Kumar, J. P. Singh, and K. Li, “The parsec benchmark suite: Characterization and architectural implications,” in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ACM, 2008, pp. 72–81.

[78] J. Edler, *Dinero IV Trace-Driven Uniprocessor Cache Simulator*. [Online]. Available: `http://pages.cs.wisc.edu/~markhill/DineroIV/`.

[79] M. Nanjundappa, V. Tech, B. A. Jose, V. Tech, and V. Tech, “SCGPSim: A fast SystemC simulator on GPUs,” in *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, 2010.

[80] N. Bombieri, S. Vinco, V. Bertacco, and D. Chatterjee, “SystemC simulation on GP-GPUs,” in *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, 2012, p. 343.

[81] W. Chen, X. Han, and R. Dömer, “Out-of-order parallel simulation for ESL design,” in *Proceedings of the Design, Automation & Test in Europe*, 2012, pp. 141–146.

[82] W. Chen and R. Doemer, “Optimized Out-of-Order Parallel Discrete Event Simulation Using Predictions,” in *Proceedings of the Design, Automation & Test in Europe*, 2013, pp. 3–8.

[83] M. Moy, “Parallel Programming with SystemC for Loosely Timed Models: A Non-Intrusive Approach,” in *Proceedings of the Design, Automation & Test in Europe*, 2013, pp. 9–14.

[84] M. Scott and W. Bolosky, “False sharing and its effect on shared memory performance,” in *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems*, 1993, p. 57.

[85] R. W. Floyd, “Algorithm 97: Shortest path,” *Communications of the ACM*, vol. 5, no. 6, p. 345, 1962.

요약

시뮬레이션은 현재의 아키텍처를 통해 미래의 아키텍처를 프로토타이핑하는 소프트웨어 기술로서 현대 컴퓨터 아키텍처 연구에서 가장 중요한 기술 중에 하나이다. 특히 새로운 아키텍처 설계 시에 시뮬레이션 기술은 복잡한 시스템의 중요한 성능 지표를 제공함으로써, 효율적인 아키텍처의 설계 공간 탐색에 사용된다. 이뿐 아니라 새로운 아키텍처가 존재하지 않는 상황에서의 소프트웨어 개발 시에도 시뮬레이션 기술은 가상의 아키텍처를 제공하여 주고, 쉽게 얻을 수 없는 다양한 통계를 제공한다. 그리하여 느린 속도와 커버리지 문제 등 여러가지 우려에도 불구하고 컴퓨터 아키텍처 연구에서의 시뮬레이션 기술의 의존도는 계속해서 증가하고 있다.

트랜지스터의 집적도가 높아지고 단일 코어의 성능 향상이 벽에 부딪힘에 따라, 최근 새로이 연구되는 아키텍처는 대부분 멀티 혹은 매니코어 아키텍처로 구성되고, 확장성 있는 통신을 위해 네트워크-온-칩 구조를 주로 사용한다. 이뿐 아니라 이러한 병렬적인 아키텍처를 효과적으로 활용하기 위해서 애플리케이션의 구현 자체도 복잡해졌다. 이에 따라 병렬 아키텍처와 병렬 애플리케이션을 위한 시뮬레이터도 굉장히 복잡해졌고, 기존에 존재하는 순차적인 시뮬레이터들로는 이러한 시스템을 더 이상 현실적인 시간에 시뮬레이션 할 수 없게 되었다.

이러한 문제를 해결하기 위한 병렬 시뮬레이션 기법들이 많이 개발되고 있으나, 너무 속도가 느리거나, 속도를 위해 정확도를 크게 희생하는 등의 문제가 있었다. 그리하여 본 논문에서는 정확도를 거의 희생하지 않는 선에서 최선의 속도를 낼 수 있는 매니-코어 시뮬레이션 기법을 제안하여 개발하고

평가하였다.

제안하고 있는 병렬 매니-코어 시뮬레이터는 크게 세 부분으로 나뉜다. 첫째는 코어 시뮬레이터이고 둘째는 네트워크 시뮬레이터 셋째는 시뮬레이션 백플레인이다. 간단히 설명하면 각각의 코어는 코어 시뮬레이터에 의해 실행되게 되고, 필요에 따라 IPC(interprocess communication)를 통해 외부의 시뮬레이션 백플레인과 통신한다. 코어는 각각 다른 호스트 프로세서에서 각각 병렬적으로 수행된다. 그리고 시뮬레이션 백플레인에서는 각 코어로부터 온 메시지들을 시간 순서대로 정렬하여 각 모듈에 전달하고 코어 이외의 다른 하드웨어 컴포넌트들을 시뮬레이션한다. 시뮬레이션 백플레인에서 NoC 통신이 필요한 요청이 있음이 확인되면, 이 요청은 네트워크 시뮬레이터로 전달되어 가장 정확한 수준인 플릿 레벨에서 시뮬레이션된다.

이를 위해 먼저 본 논문에서는 코어 시뮬레이션을 위해 분석적 시뮬레이션과 샘플 시뮬레이션을 결합한 형태의 새로운 타이밍 모델을 기반으로 한 코어 시뮬레이션 모델을 제안하였다. 가장 널리 사용되고 있는 기능 에뮬레이터 중에 하나인 QEMU 위에 개발한 기술을 구현하여, 이렇게 개발한 각 코어 시뮬레이터의 성능이 18.62 - 44.31 MIPS에 이르고 오류는 8% 가량임을 확인하였다. 그리고 개발한 코어 시뮬레이터는 독립적으로 오픈소스로 공개되었다.

한편 NoC 시뮬레이션이 본 논문에서 겨냥하고 있는 매니코어 시뮬레이터로부터 생성되는 결과의 신뢰도에 굉장히 큰 영향을 미친다는 것을 먼저 확인하였다. 기존의 플릿-레벨 NoC 시뮬레이터들을 소스-코드 레벨로 분석하여 다양한 구현을 평가하고 소프트웨어적인 최적화를 가해 속도를 향상시켰다. 그리고 각 최적화가 성능에 미치는 영향을 실험을 통해 확인하였다. 이렇게 개발된 NoC 시뮬레이터는 8x8 NoC에서 매 시뮬레이션 사이클 당 패킷 생성율이 0.00625 미만인 경우 100KCycles/s 이상의 성능을 보여주었고, 이는 기존에 존재하는

NoC 시뮬레이터들에 비해 최소 2배 이상 빠른 것이다.

시뮬레이션 백플레인의 속도는 IPC 오버헤드와 SystemC 스케줄링 오버헤드에 의해 크게 좌우된다. IPC 오버헤드를 줄이기 위해서 트레이스 기반의 코시뮬레이션 기법을 사용하고, 빠른 IPC를 도입하였으며, L1 데이터 캐시를 가상으로 분할하여 코어 시뮬레이터 포함시켰다. 그리고 SystemC 스케줄링 오버헤드 자체를 줄이기 위해 동시에 깨어나는 모듈의 수를 줄이기 위해 이벤트-기반 슬레이브 모듈 프로세싱을 사용하였고, 새로운 스케줄러 병렬화 기법을 연구하였다. 새로이 개발한 SystemC 병렬 스케줄러는 제한된 상황에서는 좋은 성능을 보여주었지만, 본 논문에서 개발한 TLM 수준의 매니코어 시뮬레이터에서는 별다른 성능 향상이 발견되지 않는다는 것도 확인하였다.

최종적으로 개발된 매니코어 시뮬레이터는 인과관계 오류를 허용하지 않는 보수적인 동기화 기법을 사용하고 플릿 수준의 가장 정확한 NoC 시뮬레이션을 함에도 불구하고 굉장히 고속이라는 차별성이 있다. 또한 SystemC TLM 2.0에 호환되게 만들어 졌기 때문에, 추후 다른 모듈을 추가하는 등의 확장성이 굉장히 뛰어나기도 하다. 정확도에 대한 실험이 많이 이루어지지는 못했지만, 이는 타겟 아키텍처에 대한 상세한 스펙이 주어졌을 때 보완 가능할 것이다.

본 박사 논문은 앞으로도 더욱 필수적이 될 매니코어 시뮬레이터 개발 시에 참고할 만한 논문이 될 수 있을 것이다.

주요어: 매니코어, 멀티코어, 네트워크-온-칩, 병렬 시뮬레이션, 가상 프로토타이핑, 동기화

학번: 2010-20747